List classes

The files **g++-include/List.hP** and **g++-include/List.ccP** provide pseudo-generic Lisp-type List classes. These lists are homogeneous lists, more similar to lists in statically typed functional languages like ML than Lisp, but support operations very similar to those found in Lisp. Any particular kind of list class may be generated via the **genclass** shell command. However, the implementation assumes that the base class supports an equality operator **==**. All equality tests use the **==** operator, and are thus equivalent to the use of **equal**, not **eq** in Lisp.

All list nodes are created dynamically, and managed via reference counts. **List** variables are actually pointers to these list nodes. Lists may also be traversed via Pixes, as described in the section describing Pixes. See *Pseudo-indexes* in **/NextLibrary/Documentation/GNU/libg++/Intro.rtf**.

Supported operations are mirrored closely after those in Lisp. Generally, operations with functional forms are constructive, functional operations, while member forms (often with the same name) are sometimes procedural, possibly destructive operations.

As with Lisp, destructive operations are supported. Programmers are allowed to change head and tail fields in any fashion, creating circular structures and the like. However, again as with Lisp, some operations implicitly assume that they are operating on pure lists, and may enter infinite loops when presented with improper lists. Also, the reference-counting storage management facility may fail to reclaim unused circularly-linked nodes.

Several Lisp-like higher order functions are supported (e.g., **map**). Typedef declarations for the required functional forms are provided int the **.h** file.

For purposes of illustration, assume the specification of class **intList**. Common Lisp versions of supported operations are shown in brackets for comparison purposes.

Constructors and assignment

intList a;	[(setq a nil)]	Declares a to be a nil intList.
intList b(2);	[(setq b (cons 2 nil))]	Declares b to be an intList with a head value of 2, and a nil tail.
intList c(3, b);	[(setq c (cons 3 b))]	Declares c to be an intList with a head value of 3, and b as its tail.
b = a;	[(setq b a)]	Sets b to be the same list as a.

Assume the declarations of intLists a, b, and c in the following. See *Pseudo-indexes* in **/NextLibrary/Documentation/GNU/libg++/Intro.rtf**.

List status

a.null(); OR !a;	[(null a)]	returns true if a is null.
a.valid();	[(listp a)]	returns true if a is non-null. Inside a conditional test, the **void*** coercion may also be used as in **if (a) ...**.
intList();	[nil]	intList() may be used to null terminate a list, as in **intList f(int x) {if (x == 0) return intList(); ... }**.
a.length();	[(length a)]	returns the length of a.
a.list_length();	[(list-length a)]	returns the length of a, or -1 if a is circular.

heads and tails

a.get(); OR **a.head()**	[(car a)]	returns a reference to the head field.
a[2];	[(elt a 2)]	returns a reference to the second (counting from zero) head field.
a.tail();	[(cdr a)]	returns the intList that is the tail of a.
a.last();	[(last a)]	returns the intList that is the last node of a.
a.nth(2);	[(nth a 2)]	returns the intList that is the nth node of a.
a.set_tail(b);	[(rplacd a b)]	sets a's tail to b.
a.push(2);	[(push 2 a)]	equivalent to a = intList(2, a);
int x = a.pop()	[(setq x (car a)) (pop a)]	returns the head of a, also setting a to its tail.

Constructive operations

b = copy(a);	[(setq b (copy-seq a))]	sets b to a copy of a.
b = reverse(a);	[(setq b (reverse a))]	Sets b to a reversed copy of a.
c = concat(a, b);	[(setq c (concat a b))]	Sets c to a concatenated copy of a and b.

c = append(a, b); [(setq c (append a b))] Sets c to a concatenated copy of a and b. All nodes of a are copied, with the last node pointing to b.

b = map(f, a); [(setq b (mapcar f a))] Sets b to a new list created by applying function f to each node of a.

c = combine(f, a, b); Sets c to a new list created by applying function f to successive pairs of a and b. The resulting list has length the shorter of a and b.

b = remove(x, a); [(setq b (remove x a))] Sets b to a copy of a, omitting all occurrences of x.

b = remove(f, a); [(setq b (remove-if f a))] Sets b to a copy of a, omitting values causing function f to return true.

b = select(f, a); [(setq b (remove-if-not f a))] Sets b to a copy of a, omitting values causing function f to return false.

c = merge(a, b, f); [(setq c (merge a b f))] Sets c to a list containing the ordered elements (using the comparison function f) of the sorted lists a and b.

Destructive operations

a.append(b); [(rplacd (last a) b)] appends b to the end of a. No new nodes are constructed.

a.prepend(b); [(setq a (append b a))] prepends b to the beginning of a.

a.del(x); [(delete x a)] deletes all nodes with value x from a.

a.del(f); [(delete-if f a)] deletes all nodes causing function f to return true.

a.select(f); [(delete-if-not f a)] deletes all nodes causing function f to return false.

a.reverse(); [(nreverse a)] reverses a in-place.

a.sort(f); [(sort a f)] sorts a in-place using ordering (comparison) function f.

a.apply(f); [(mapc f a)] Applies void function f (int x) to each element of a.

a.subst(int old, int repl); [(nsubst repl old a)] substitutes repl for each occurrence of old in a. Note the different argument order than the Lisp version.

Other operations

a.find(int x); [(find x a)] returns the intList at the first occurrence of x.

a.find(b); [(find b a)] returns the intList at the first occurrence of sublist b.

a.contains(int x); [(member x a)] returns true if a contains x.

a.contains(b); [(member b a)] returns true if a contains sublist b.

a.position(int x); [(position x a)] returns the zero-based index of x in a, or -1 if x does not occur.

int x = a.reduce(f, int base); [(reduce f a :initial-value base)] Accumulates the result of applying int function

f(int, int) to successive elements of a, starting with base.