

Implementing lists

Want: null, hd, tl, ::

How: Arrays? Pointers? Other?

Costs: memory, time, code

CSE 341 Spring 2006, Lecture 6

2

CSE 341: Programming Languages

Spring 2006

Lecture 6 — More on Tail Recursion & Accumulators

CSE 341 Spring 2006, Lecture 6

1

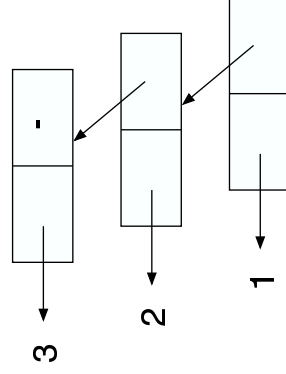
Implementing lists

Want: null, hd, tl, ::

How: Arrays? Pointers? Other?

Costs: memory, time, code

[1,2,3]



CSE 341 Spring 2006, Lecture 6

3

Using Lists (Java)

Consider a linked list of integers, implemented in Java.

How would you implement functions for:

- Finding the *length* of a list
- Finding the *last element* of a list

CSE 341 Spring 2006, Lecture 6

4

Using Lists (ML)

Consider

```
fun len [] = 0
  | len (x::xs) = 1 + len xs;

val theLength = len [1,2,3,4,5];
```

Q: How do you implement function call?

A: A "Call Stack"

CSE 341 Spring 2006, Lecture 6

5

Implementing calls

Consider

```
fun len [] = 0
  | len (x::xs) = 1 + len xs;

val theLength = len [1,2,3,4,5];

Compare:

fun last [x] = x
  | last(x::xs) = last xs;

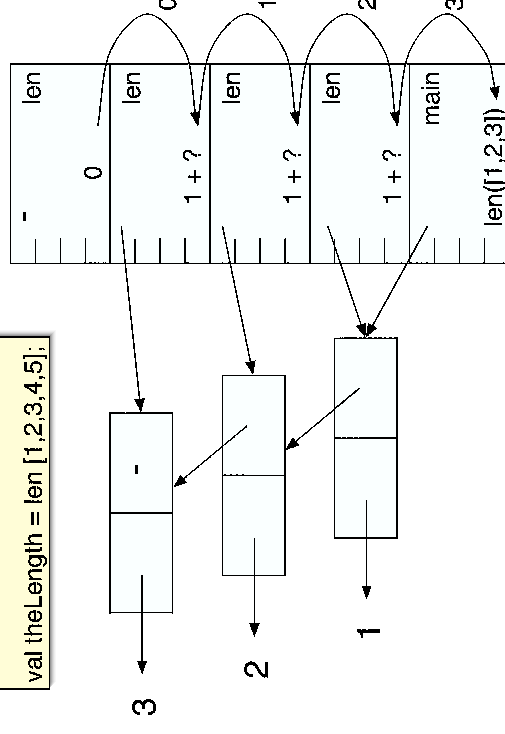
val theLast = last [1,2,3,4,5];
```

CSE 341 Spring 2006, Lecture 6

7

```
fun len [] = 0
  | len (x::xs) = 1 + len xs;

val theLength = len [1,2,3,4,5];
```



CSE 341 Spring 2006, Lecture 6

6

Tail calls

If the result of $f(x)$ is the result of the enclosing function body, then $f(x)$ is a *tail call*.

More precisely, a tail call is a call in *tail position*:

- In `fun f(x) = e`, `e` is in tail position.
- If `if e1 then e2 else e3` is in tail position, then `e2` and `e3` are in tail position (not `e1`). (Similar for `case`).
- If `let b1 ... bn in e` is in tail position, then `e` is in tail position (not any binding expressions).
- Function arguments are not in tail position.
- ...

CSE 341 Spring 2006, Lecture 6

8

So what?

Why does this matter?

- Implementation takes space proportional to depth of function calls (“call stack” must “remember what to do next”)
- But in functional languages, implementation must ensure tail calls eliminate the caller’s space
- Accumulators are a systematic way to make some functions tail recursive
- “Self” tail-recursive is very loop-like because space does not grow.