Currying

• Currying (after Haskell Curry)
  – technique of transforming a function that takes multiple arguments (an n-tuple) in such a way as it can be called as a chain of functions each with a single argument
Example

- fun prod(a,b) = a * b;
- Prod: : int * int -> int

- val cprod = curry prod;
- val cprod = fn : int -> int -> int

- cprod 6;
- val it = fn : int -> int
curry & uncurry

• val curry = fn f => fn x => fn y => f(x, y);
• curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c

• val uncurry = fn f => fn (x, y) => f x y;
• uncarry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
Insertion sort
(non-iterative is faster)

- fun ins (x, []): real list = [x]
-   | ins (x, y::ys) =
-     if x<=y then x::y::ys (*it belongs here*)
-     else y::ins(x,ys);

- fun insort [] = []
-   | insort (x::xs) = ins(x, insort xs);
Insertion sort
(using \textit{let})

\begin{itemize}
  \item \texttt{fun insort lessequal =}
  \item \texttt{\hspace{1cm} let fun ins (x, []) = [x]}
  \item \texttt{\hspace{1cm} \hspace{1cm} | ins (x, y::ys) =}
  \item \texttt{\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} if lessequal(x,y) then x::y::ys}
  \item \texttt{\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} else y :: ins (x,ys)}
  \item \texttt{\hspace{1cm} fun sort [] = []}
  \item \texttt{\hspace{1cm} \hspace{1cm} | sort (x::xs) = ins (x, sort xs)}
  \item \texttt{\hspace{1cm} in \hspace{1cm} sort \hspace{1cm} end;}
\end{itemize}
Graph algorithms

• Representation:
  – List of pairs

• We need a function to explore the arcs:

• fun nexts (a, []) = []
•     | nexts (a, (x,y)::pairs) =
•         if a=x then  y :: nexts(a,pairs)
•         else       nexts(a,pairs);
depth-first search

• fun depthf ([], graph, visited) = rev visited
  • | depthf (x::xs, graph, visited) =
    • if x mem visited then depthf (xs, graph, visited)
    • else depthf (nexts(x,graph) @ xs, graph, x::visited);

• Test on the following graph:
• [(1,2),(1,3),(2,3),(2,5),(2,4),(4,5),(4,1)]
Lambda expression datatype

- type var = string

- datatype lexp = LAMBDA of var * lexp
  - | APP of lexp * lexp
  - | VAR of var

- $(\lambda x . (\lambda y . x)) x$

- APP (LAMBDA (x,(LAMBDA (y,x))))) x;
freevars function

val freevars =
  let fun freevars' fvs (VAR v) = if List.exists (fn v' => v = v') fvs
      then fvs
      else v :: fvs
    | freevars' fvs (LAMBDA (v, e)) = let val fvs' = freevars' [] e
        in fvs @ (remove v fvs')
        end
    | freevars' fvs (APP (e1, e2)) = freevars' (freevars' fvs e1) e2
      in freevars' []
    end