Accumulators

- Accumulator programming is a way to handle state in declarative programs. It is a programming technique that uses arguments to carry state, transform the state, and pass it to the next procedure.
- Assume that the state $S$ consists of a number of components to be transformed individually: $S = (X_1, Y_1, Z_1,...)$
- For each predicate $P$, each state component is made into a pair, the first component is the input state and the second component is the output state after $P$ has terminated
- $S$ is represented as $(X_{out}, Y_{out}, Z_{out}, Z_{out},...)$

MergeSort Example

- Consider a variant of MergeSort with accumulator
- $\text{proc} \{ \text{MergeSort1} \ N \ S0 \ S \ Xs \}$
  - $N$ is an integer,
  - $S0$ is an input list to be sorted
  - $S$ is the remainder of $S0$ after the first $N$ elements are sorted
  - $Xs$ is the sorted first $N$ elements of $S0$
- The pair $(S0, S)$ is an accumulator
- The definition is in a procedural syntax in Oz because it has two outputs $S$ and $Xs$
Example (2)

```prolog
fun {MergeSort Xs} {MergeSort1 {Length Xs} Xs _ Ys} Ys end
proc {MergeSort1 N S0 S Xs} if N==0 then S = S0 Xs = nil elseif N ==1 then X in X|S = S0 Xs=[X] else %% N > 1 local S1 Xs1 Xs2 NL NR in NL = N div 2 NR = N - NL {MergeSort1 NL S0 S1 Xs1} {MergeSort1 NR S1 S Xs2} Xs = {Merge Xs1 Xs2} end end end
```

MergeSort Example in Prolog

```prolog
mergesort(Xs,Ys) :- length(Xs,N), mergesort1(N,Xs,_ , Ys). mergesort1(0,S,S,[] :- !. mergesort1(1,[X|S],S,[X]) :- !. mergesort1(N,S0,S,Xs):- NL is N // 2, NR is N - NL, mergesort1(NL,S0,S1,Xs1), mergesort1(NR,S1,S,Xs2), merge(Xs1,Xs2,Xs).```

Multiple accumulators

- Consider a stack machine for evaluating arithmetic expressions
- Example: (1+4)-3
- The machine executes the following instructions
  - push(1)
  - push(4)
  - plus
  - push(3)
  - minus

```
| 1 | 4 | 5 | 3 | 2 |
```

Multiple accumulators (2)

- Example: (1+4)-3
- The arithmetic expressions are represented as trees: minus(plus(1 4) 3)
- Write a procedure that takes arithmetic expressions represented as trees and output a list of stack machine instructions and counts the number of instructions

```prolog
proc {ExprCode Expr Cin Cout Nin Nout} case Expr of plus(Expr1 Expr2) then C1 N1 in C1 = plus|C0 N1 = N0 + 1 {SeqCode [Expr2 Expr1] C1 C N1 N} minus(Expr1 Expr2) then C1 N1 in C1 = minus|C0 N1 = N0 + 1 {SeqCode [Expr2 Expr1] C1 C N1 N} I andthen {IsInt I} then C = push(I)|C0 N = N0 + 1 end proc {SeqCode Es C0 C N0 N} case Es of nil then C = C0 N = N0 E|Er then N1 C1 in {ExprCode E C0 C1 N1 N0} {SeqCode Er C1 C N1 N} end end```

Multiple accumulators (3)

```
proc {ExprCode Expr C0 C N0 N} case Expr of plus(Expr1 Expr2) then C1 N1 in C1 = plus|C0 N1 = N0 + 1 {SeqCode [Expr2 Expr1] C1 C N1 N} minus(Expr1 Expr2) then C1 N1 in C1 = minus|C0 N1 = N0 + 1 {SeqCode [Expr2 Expr1] C1 C N1 N} I andthen {IsInt I} then C = push(I)|C0 N = N0 + 1 end end
```

Multiple accumulators (4)

```
proc {ExprCode Expr C0 C N0 N} case Expr of plus(Expr1 Expr2) then C1 N1 in C1 = plus|C0 N1 = N0 + 1 {SeqCode [Expr2 Expr1] C1 C N1 N} minus(Expr1 Expr2) then C1 N1 in C1 = minus|C0 N1 = N0 + 1 {SeqCode [Expr2 Expr1] C1 C N1 N} I andthen {IsInt I} then C = push(I)|C0 N = N0 + 1 end end
```
Difference lists in Oz

- A difference list is a pair of lists, each might have an unbound tail, with the invariant that one can get the second list by removing zero or more elements from the first list
  - X ≠ X % Represent the empty list
  - nil ≠ nil % idem
  - [a] ≠ [a] % idem
  - (a/b/c/X) ≠ X % Represents [a b c]
  - [a b c d] ≠ [d] % idem

Difference lists in Prolog

- A difference list is a pair of lists, each might have an unbound tail, with the invariant that one can get the second list by removing zero or more elements from the first list
  - X , X % Represent the empty list
  - [] , [] % idem
  - [a] , [a] % idem
  - [a,b,c,X] , X % Represents [a,b,c]
  - [a,b,c,d] , [d] % idem

Difference lists in Oz (2)

- When the second list is unbound, an append operation with another difference list takes constant time
  - fun {AppendD D1 D2} S1 # E1 = D1 S2 # E2 = D2 in E1 = S2 S1 # E2 end
  - local X Y in {Browse {AppendD (1|2|3|X)#X (4|5|Y)#Y}} end
  - Displays [1|2|3|4|5|Y]#Y

Difference lists in Prolog (2)

- When the second list is unbound, an append operation with another difference list takes constant time
  - append_d(S1,E1, S2,E2) :- E1 = S2.
A FIFO queue with difference lists (1)

- A FIFO queue is a sequence of elements with an insert and a delete operation.
  - Insert adds an element to one end and delete removes it from the other end.
- Queues can be implemented with lists. If L represents the queue content, then inserting X gives X|L and deleting X gives {ButLast L X} (all elements but the last).
  - Delete is inefficient: it takes time proportional to the number of queue elements.
- With difference lists we can implement a queue with constant-time insert and delete operations.
  - The queue content is represented as q(N S E), where N is the number of elements and S#E is a difference list representing the elements.

A FIFO queue with difference lists (2)

fun (NewQueue) X in q(0 X X) end

fun (Insert Q X)
  case Q of q(N S E) then E1 in
  X=X|E1 q(N+1 S E1) end
end

fun (Delete Q X)
  case Q of q(N S E) then
  S1 in
  X|S=S1 q(N-1 S1 E) end
end

fun (EmptyQueue)
  case Q of q(N S E) then
  N==0 end end

fun (Flatten Xs)
  case Xs of
  nil then
  nil
  [] X|Xr andthen {IsLeaf X} then
  X|{Flatten Xr}
  [] X|Xr andthen {Not {IsLeaf X}} then
  {Append {Flatten X} {Flatten Xr}} end
end

fun (FlattenD Xs Ds)
  case Xs of
  nil then
  Y in Ds = Y#Y
  [] X|Xr andthen Y0 Y1 Y2 in
  Ds = Y0#Y2
  (FlattenD X Y0#Y1)
  (FlattenD X Y1#Y2)
  [] X andthen {IsLeaf X} then
  Y in (X|Y)#Y end
end

fun (Flatten Xs) Y in (FlattenD Xs Y#nil) Y end

Fun (Reverse Xs)

fun (Reverse Xs)
  case Xs of
  nil then nil
  [] X|Xr then (Append (Reverse Xr) X) end
end

fun (Reverse nil) X in Y#X end

Reverse with difference lists

- Here is our recursive reverse:
  - Inserting "b":
    - In q(2 a|b|U U) and q(2 a|b|U U) and q(1 b|U U)
  - Deleting "X":
    - In q(1 b|U U) and q(1 b|U U)
  - Difference list allows operations at both ends.
  - N is needed to keep track of the number of queue elements.

Flatten with difference lists (1)

- Flatten of nil is X#X
- Flatten of X|Xs is Y1#Y where
  - Flatten of Xs is Y1#Y2
  - Flatten of X is Y3#Y
  - equate Y2 and Y3
- Flatten of a leaf X is (X|Y)#Y

Flatten with difference lists (2)

proc (FlattenD Xs Ys Zs)
  case Xs of
  nil then
  Y in Zs = Y#Y
  [] X|Xr andthen Y0 Y1 Y2 in
  Zs = Y0#Y2
  (FlattenD X Y0#Y1)
  (FlattenD X Y1#Y2)
  [] X andthen {IsLeaf X} then
  Y in (X|Y)#Y end
end

Let us replace lists by difference lists and see what happens.
Reverse with difference lists (1)

- The naive version takes time proportional to the square of the input length.
- Using difference lists in the naive version makes it linear time.
- We use two arguments $Y_1$ and $Y$ instead of $Y_1#Y$.
- With a minor change we can make it iterative as well.

```haskell
fun {ReverseD Xs}
proc {ReverseD Xs Y1 Y}
  case Xs
  of nil then Y1=Y
  [] X|Xr then Y2 in
    {ReverseD Xr Y1 Y2}
  end
  end
  end
  in
  {ReverseD Xs R nil}
  R
end
end
```

Reverse with difference lists (2)

```haskell
fun {ReverseD Xs}
proc {ReverseD Xs Y1 Y}
  case Xs
  of nil then Y1=Y
  [] X|Xr then
    {ReverseD Xr Y1 X|Y}
  end
  end
  R in
  {ReverseD Xs R nil}
  R
end
```

Difference lists: Summary

- Difference lists are a way to represent lists in the declarative model such that one append operation can be done in constant time.
  - A function that builds a big list by concatenating together lots of little lists can usually be written efficiently with difference lists.
  - The function can be written naively, using difference lists and append, and will be efficient when the append is expanded out.
- Difference lists are declarative, yet have some of the power of destructive assignment.
  - Because of the single-assignment property of dataflow variables.
- Difference lists originated from Prolog and are used to implement, e.g., definite clause grammar rules for natural language parsing.

Exercises

15. Draw the search trees for Prolog queries:
   - `append([1,2],[3],[L]).`
   - `append(X,Y,[1,2,3]).`
   - `append_di([1,2]|X|X,[3|Y],[Y,S,E]).`
16. Rewrite the multiple accumulators example in Prolog.
17. VRH Exercise 3.10.11 (page 232)
18. VRH Exercise 3.10.14 (page 232)
19. VRH Exercise 3.10.15 (page 232)