Higher-order programming

- Defining a procedure as a procedure value with a contextual environment is enormously expressive
  - It is arguably the most important invention in programming languages: it makes possible building large systems based on data abstraction
- Since procedures (and functions) are values, we can pass them as inputs to other functions and return them as outputs
  - Remember that in our kernel language, we consider functions and procedures to be the same concept: a function is a procedure with an extra output argument

Order of a function

- We define the order of a function (or procedure)
  - A function whose inputs and output are not functions is first order
  - A function is order N+1 if its inputs and output contain a function of maximum order N
- Let's give some examples to show what we can do with higher-order functions (where the order is greater than 1)
  - We will give more examples later in the course

Genericity

- Genericity is when a function is passed as an input

```
declare
fun {Map F L}
      case L of nil then nil
      [] H|T then {F H}|{Map F T}
      end
end

{Browse {Map fun {$ X} X*X end [7 8 9]}}
```

What is the order of Map in this call?

Instantiation

- Instantiation is when a function is returned as an output

```
declare
fun {MakeAdd A}
    fun {$ X} X+A end
end
Add5={MakeAdd 5}

{Browse {Add5 100}}
```

What is the order of MakeAdd?

What is the contextual environment of the function returned by MakeAdd?

Function composition

- We take two functions as input and return their composition

```
declare
fun {Compose F G}
    fun {$ X} {F {G X}} end
end
Fnew={Compose fun {$ X} X*X end
              fun {$ X} X+1 end}
```

What is the contextual environment of the function returned by Compose?

- What does {Fnew 2} return?
- What does {{Compose Fnew Fnew} 2} return?

Abstracting an accumulator

- We can use higher-order programming to do a computation that hides an accumulator
- Let's say we want to sum the elements of a list $L=[a_0\ a_1\ a_2\ \ldots\ a_{n-1}]$:
  - $S = a_0 + a_1 + a_2 + \ldots + a_{n-1}$
  - $S = (\ldots(((0 + a_0) + a_1) + a_2) + \ldots + a_{n-1})$
- We can write this generically with a function F:
  - S = {F … {F {F {F 0 a_0} a_1} a_2} … a_{n-1}}
- Now we can define the higher-order function FoldL:
  - S = {FoldL [a_0 a_1 a_2 … a_{n-1}] F 0}
  - The accumulator is hidden inside FoldL!

Definition of FoldL

- Here is the definition of FoldL:

```
declare
fun {FoldL L F U}
    case L
    of nil then U
    [] H|T then {FoldL T F {F U H}}
    end
end
S={FoldL [5 6 7] fun {$ X Y} X+Y end 0}
```

The argument U is an accumulator

Encapsulation

- We can hide a value inside a function:

```
declare
fun {Zero} 0 end
fun {Inc H}
N={H}+1 in
      fun {$} N end
end
Three={Inc {Inc {Inc Zero}}}
{Browse {Three}}
```

- This is the foundation of encapsulation as used in data abstraction
- What is the difference if we write Inc as follows:

```
fun {Inc H} fun {$} {H}+1 end end
```

Delayed execution

- We can define an statement and pass it to a function which decides whether or not to execute it

```
proc {IfTrue Cond Stmt}
     if {Cond} then {Stmt} end
end
Stmt = proc {$} {Browse 111*111} end
{IfTrue fun {$} 1<2 end Stmt}
```

- This can be used to build control structures from scratch (**if** statement, **while** loop, **for** loop, etc.)

Summary of higher-order

- We have given six examples to illustrate the expressiveness of higher-order programming:
  - Genericity
  - Instantiation
  - Function composition
  - Abstracting an accumulator
  - Encapsulation
  - Delayed execution

- We will use these techniques and others when we introduce the concepts of data abstraction
  - Data abstraction is built on top of higher-order programming!