**1. *A calculator*. Section 1.1 uses the system as a calculator. Let us explore the possibilities:**

**(a) Calculate the exact value of 2100 without using any new functions. Try to think of shortcuts to do it without having to type 2\*2\*2\*...\*2 with one hundred 2s. *Hint*: use variables to store intermediate results.**

*We use the idea that 2100 = (210)10*

declare V = 2\*2\*2\*2\*2\*2\*2\*2\*2\*2

{Browse V\*V\*V\*V\*V\*V\*V\*V\*V\*V}

*The browser then displays 1 267 650 600 228 229 401 496 703 205 376 which can be proven correct using a calculator.*

**(b) Calculate the exact value of 100! without using any new functions. Are there any possible shortcuts in this case?**

*No, there are no possible shortcuts in this case.*

**2. *Calculating combinations*. Section 1.3 defines the function Comb to calculate combinations. This function is not very efficient because it might require calculating very large factorials. The purpose of this exercise is to write a more efficient version of Comb.**

**(a) As a first step, use the following alternative definition to write a more efficient function:**

declare Comb2 Numerator Denominator Iterator Fact

fun {Comb2 N K}

if K == 0 then 1

else

{Numerator N K} div {Denominator K} end

end

fun {Num N K}

local S=N-K+1 in

fun {Iterator N A}

if N<S then A

else {Iterator N-1 N\*A}

end

end

{Iterator N 1}

end

end

declare

fun {Fact N}

if N==0 then 1 else N\*{Fact N-1} end

end

fun {Den K}

{Fact K}

end

{Browse {Comb2 10 3}}

*The above call displays 120, which is correct.*

**Calculate the numerator and denominator separately and then divide them.**

**Make sure that the result is 1 when *k* = 0.**

**(\_b) As a second step, use the following identity:**

***n***

***k***

**\_**

**=**

**\_**

***n***

***n − k***

**\_**

**to increase efficiency even more.That is, if *k > n/*2, then do the calculation**

**with *n − k* instead of with *k*.**

*The only difference between Comb2 above and Comb3 below is that Comb3 has an additional clause to account for the identity (also shown above).*

declare Comb3 Numerator Denominator Iterator Factorial

fun {Comb3 N K}

if K == 0 then 1

elseif K > (N div 2) then {Comb3 N N-K}

else

{Numerator N K} div {Denominator K} end

end

fun {Numerator N K}

local S=N-K+1 in

fun {Iterator N A}

if N<S then A

else {Iterator N-1 N\*A}

end

end

{Iterator N 1}

end

end

fun {Factorial N}

if N==0 then 1 else N\*{Factorial N-1} end

end

fun {Denominator K}

{Factorial K}

end

3. *Program correctness*. Section 1.6 explains the basic ideas of program correctness and applies them to show that the factorial function defined in section 1.3 is correct.

In this exercise, apply the same ideas to the function Pascal of section 1.5 to show that it is correct.

**4. *Program complexity*. What does section 1.7 say about programs whose time complexity is a high-order polynomial? Are they practical or not? What do you think?**

*Section 1.7 does not say anything about programs whose time complexity is high-order polynomial, although it is mentioned that “programs whose time complexity is exponential are impractical except for very small inputs”. Since high-order polynomial functions behave in a similar way to exponential functions with regards to their rates of growth, programs whose time complexity is high-order polynomial are just as impractical as those whose complexity is exponential since the execution time will blow up at similar rates in both cases.*

**5. *Lazy evaluation*. Section 1.8 defines the lazy function *Ints* that lazily calculates an infinite list of integers. Let us define a function that calculates the sum of a list of integers:**

**fun** {SumList L}

**case** L **of** X|L1 **then** X+{SumList L1}

**else** 0 **end**

**end**

**What happens if we call {SumList {Ints 0}}? Is this a good idea?**

The call to {SumList {Ints 0}} does not produce any result, this is because Ints is a lazy function, meaning that function values for Ints are not evaluated unless they are needed. Since SumList needs the result of {Ints 0} in order to calculate a result, it does not execute.

6. *Higher-order programming*. Section 1.9 explains how to use higher-order programming to calculate variations on Pascal’s triangle. The purpose of this exercise is to explore these variations.

(a) Calculate individual rows using subtraction, multiplication, and other operations. Why does using multiplication give a triangle with all zeros? Try the following kind of multiplication instead:

**fun** {Mul1 X Y} (X+1)\*(Y+1) **end**

What does the 10th row look like when calculated with Mul1?

(b) The following loop instruction will calculate and display 10 rows at a time:

**for** I **in** 1..10 **do** {Browse {GenericPascal Op I}} **end**

Use this loop instruction to make it easier to explore the variations.

7. *Explicit state*.This exercise compares variables and cells. We give two code

fragments.Th e first uses variables:

**local** X **in**

X=23

**local** X **in**

X=44

**end**

{Browse X}

**end**

The second uses a cell:

**local** X **in**

X={NewCell 23}

X:=44

{Browse @X}

**end**

In the first, the identifier X refers to two different variables.In the second, X refers

to a cell.What does Browse display in each fragment? Explain.

8. *Explicit state and functions*.This exercise investigates how to use cells together

with functions.Let us define a function {Accumulate N} that accumulates all its

inputs, i.e., it adds together all the arguments of all calls. Here is an example:

{Browse {Accumulate 5}}

{Browse {Accumulate 100}}

{Browse {Accumulate 45}}

*1.18 Exercises 25*

This should display 5, 105, and 150, assuming that the accumulator contains zero

at the start.H ere is a wrong way to write Accumulate:

**declare**

**fun** {Accumulate N}

Acc **in**

Acc={NewCell 0}

Acc:=@Acc+N

@Acc

**end**

What is wrong with this definition? How would you correct it?

9. *Memory store*.Th is exercise investigates another way of introducing state: a

memory store.Th e memory store can be used to make an improved version of

FastPascal that remembers previously calculated rows.

(a) A memory store is similar to the memory of a computer.It has a series

of memory cells, numbered from 1 up to the maximum used so far.Th ere are

four functions on memory stores: NewStore creates a new store, Put puts a

new value in a memory cell, Get gets the current value stored in a memory

cell, and Size gives the highest-numbered cell used so far.F or example:

**declare**

S={NewStore}

{Put S 2 [22 33]}

{Browse {Get S 2}}

{Browse {Size S}}

This stores [22 33] in memory cell 2, displays [22 33], and then displays 2.

Load into the Mozart system the memory store as defined in the supplements

file on the book’s Web site.Then use the interactive interface to understand

how the store works.

(b) Now use the memory store to write an improved version of FastPascal,

called FasterPascal, that remembers previously calculated rows.If a call asks

for one of these rows, then the function can return it directly without having

to recalculate it.This technique is sometimes called memoization since the

function makes a “memo” of its previous work.Th is improves its performance.

Here’s how it works:

First make a store S available to FasterPascal.

For the call {FasterPascal N}, let *m* be the number of rows stored in

S, i.e., rows 1 up to *m* are in S.

If *n > m*, then compute rows *m*+ 1 up to *n* and store them in S.

Return the nth row by looking it up in S.

Viewed from the outside, FasterPascal behaves identically to FastPascal

except that it is faster.

(c) We have given the memory store as a library.It turns out that the memory

store can be defined by using a memory cell.W e outline how it can be done

and you can write the definitions.The cell holds the store contents as a list of

*26 Introduction to Programming Concepts*

the form [N1|X1 ... Nn|Xn], where the cons Ni|Xi means that cell number

Ni has content Xi.Th is means that memory stores, while they are convenient,

do not introduce any additional expressive power over memory cells.

(d) Section 1.13 defines a counter object. Change your implementation of the

memory store so that it uses this counter to keep track of the store’s size.

10. *Explicit state and concurrency*.Sectio n 1.15 gives an example using a cell to

store a counter that is incremented by two threads.

(a) Try executing this example several times.W hat results do you get? Do

you ever get the result 1? Why could this be?

(b) Modify the example by adding calls to Delay in each thread.Th is changes

the thread interleaving without changing what calculations the thread does.

Can you devise a scheme that always results in 1?

(c) Section 1.16 gives a version of the counter that never gives the result 1.

What happens if you use the delay technique to try to get a 1 anyway?