A Gentle Introduction to Computer Science

Sebastian Danicic

November 11, 2010

Contents

Chapter 1

Logging on to Igor

Igor is the departmental server. it runs Linux. Linux is a version of an operating system called Unix. Apple Mac also use a version of Unix.

1.1 Logging on to Igor from a Mac

1. First fire up a terminal. This is found under *utilities*, which is found under *applications*.
2. Inside the terminal type:

   `ssh -X <username>@igor.gold.ac.uk`

   Where `<username>` is your username. Then type in your password when asked.

1.2 Logging on to Igor from Microsoft Windows

1. First fire up cygwin (or some other SSH client if you have not got cygwin).
2. Inside the terminal type:

   `startxwin`

   A new white xterm opens. Then type:

   `ssh -X <username>@igor.gold.ac.uk`

   Where `<username>` is your username. Then type in your password when asked.

Having logged on to igor you will see soemthing like:

```
---------------GOLDSMITHS COLLEGE DEPARTMENT OF COMPUTING---------------
                          (_) ___   ___  _ __
                          | |/ _ \ / _ \| '__|
                          | | (_) | (_) | |
                          |_|\__  |\___/|_|
                             |___/
    Email enquiries regarding this server to sysadmin@doc.gold.ac.uk
------------------------------------------------------------------------
[mas01sd@igor ~]$
```

1.3 Useful but not essential information

1.3.1 Copying files from a Mac or PC to igor

`scp fred.hop <username>@igor.gold.ac.uk:`

will copy file a file called `fred.hop`from your Mac or PC to igor, and

`scp  <username>@igor.gold.ac.uk:fred.hop .`

will copy file a file called `fred.hop`from igor to your Mac or PC.

1.4 Using nedit on igor

nedit is a simple text editor that allows you to type in text. We will use nedit to create Hope programs. `[mas01sd@igor ~]$` is called the operating system prompt. Type

`nedit one.hop&`

at the operating system prompt. An edit window will appear. Now type

`1+1;`

into this edit window and then save the file using the file menu (or by pressing control s).

1.4.1 Running Hope on Igor

Then type

`hope < one.hop`

Hope will then run the program in `one.hop`. and output

`>> 1 : num`

Now change `one.hop` to a different program. Save it and run it by typing

`hope < one.hop`

again.

1.4.2 Another way Running Hope on Igor

Simply type:

```
hope
```

Now you can type Hope expressions directly into the interpreter. Hpe executes them straight away. Type control d when you have finished. The disadvantage of doing it this way is you cannnot save your work.

Chapter 2

Values and Types

2.1 Numbers

Type 1; into the Hope interpreter like this:

```
>: 1;
```

The Hope system responds with

```
>> 1 : num
```

This tells us the result of evaluating the expression 1 is 1 and the **type** of the expression is `num`. i.e. a number (usually called an integer).

Now try the following

```
>: 1+1;
>: 1+2*10;
>: (1+2)*10;
```

The replies given by Hope are:

```
>> 2 : num
>> 21 : num
>> 30 : num
```

So every expression is evaluated and the type of each expression is given.

2.1.1 Exercises

1. Get Hope to work out the sum of the first 10 numbers 1+2+ ... + 10. There is no shortcut (yet!).
2. Get Hope to work out the multiple of the first 10 numbers 1*2* ... * 10. This is called 10 factorial - written 10!

3. There are two operators on integers called `mod` and `div`. Investigate their behaviour! Describe them. (Try inputting, for example, `12 mod 5` and `16 div 5`.)

2.2 Booleans

Type

```
>: 1<2;
```

into Hope. Hope replies:

```
>> true : bool
```

This means the value of `1<2` is `true` and the type of `1<2` is `bool` which is short for **boolean**.

Now try:

```
>: 1>2;
```

Hope replies:

```
>> false : bool
```

This means the value of `1>2` is `false` and the type of `1>2` is **boolean**.

The values `true` and `false` are the *only* values whose type is boolean. Every boolean expression evaluates to either `true` or `false`.

2.2.1 Boolean operators

There are three built-in boolean operators in Hope: `not`, `and`, and `or`. Now try:

```
>: not true;
```

Hope replies:

```
>> false : bool
```

and now try:

```
>: not false;
```

Hope replies:

```
>> true : bool
```

The operator `not` is a unary boolean operator since it is applied to a single boolean expression. Try:

```
>: not (not false);
```

Hope replies:

```
>> false : bool
```

Why is this? In order to work out `not (not false)` hope first evaluates the inner `(not false)` to give `true` so the whole expression reduces to `not (true)` which evaluates to `false`.

In shorthand we can write:
`not (not false)` → `not (true)`→ `false`.
Similarly: `not (not (1<2))` → `not (not (true))` → `not (false)`→ `true`.

2.2.2 Exercises

Evaluate by hand as a above and check your answers in Hope:

1. `1=2`
2. `not (not (1=2))`
3. `not (not (not (false)))`
4. `not(not (not (not (false))))-`

What do you think the rule is for `not`s?

Now try:

```
>: (1<2) and (3<10);
```

Hope replies:

```
>> true : bool
```

This is because *both* `1<2` and `3<10` evaluate to `true`.

Now try:

```
>: (1<2) and (3>10);
```

Hope replies:

```
>> false : bool
```

This is because `1<2` and `3>10` do *not* both evaluate to `true`.

Similarly now try:

```
>: (true) and (false);
```

Hope replies:

```
>> false : bool
```

This is because `true` and `false` do *not* both evaluate to `true`.

The boolean expression `false` evaluates to `false`! The operator `and` is called a binary boolean operator because it takes two boolean expressions (one on its left and the other on its right).

2.2.3 Exercises

Evaluate by hand and check your answers in Hope:

1. `true and true`
2. `true and false`
3. `false and true`
4. `false and false`

We can represent `and` in the following truth table:

A	B	A and B
T	T	T
T	F	F
F	T	F
F	F	F

2.2.4 Exercise

There is another binary boolean operate called `or`. Investigate its behaviour and hence draw its truth table.

2.3 Combining Boolean operators

2.3.1 Exercises

Evaluate by hand and check your answers in Hope:

1. `true and (true or false)`
2. `not(true and (1<2))`
3. `false or true or (7=5)`
4. `not(not(false) and false)`
5. `(true or false) and false;`
6. `true or (false and false);`

Explain the difference between the last two. What happens if we leave out the brackets? Why?

Chapter 3

Conditional Expressions

3.1 Simple Conditional Expressions

Try:

```
if 1<2
then 5
else 6;
```

Hope replies:

```
>> 5 : num
```

Now try:

```
if 1>2
then 5
else 6;
```

This time Hope replies:

```
>> 6 : num
```

These are *conditional expressions*. Hope first evaluates the boolean expression after the key word `if`. If it evaluates to `true`, Hope goes on to evaluate the `then` expression. If it evaluates to `false`, then Hope evaluates the `else` expression.

3.1.1 Exercise

Evaluate the following conditional expressions:

1. ```
   if 1>2
   then true
   else false;
   ```
   ```

2.
```
if true
then 25
else 17;
```

3.
```
if (1>2) and (2<3)
then 1+4
else 17*3;
```

4.
```
if (1>2) or not(2<3)
then 1+4
else 17*3;
```

3.1.2 Errors

The `then` and `else` part of a conditional expression must be of the same type. try:

```
if (1>2) or not(2<3)
then 1
else true;
```

Hope replies:

```
type error - conflict between branches of conditional
```

The `then` part of a conditional expression is of type `num` and the `else` part of a conditional expression is of type `bool`.

3.2 Combining Expressions into More Complicated Ones

A key idea in computer science is to combine simple things to make more complicated ones. Expressions can be combined in all sorts of ways provided the types are right: For example:

```
(if (1>2) or not(2<3)
then 1
else 2) +17;
```

Gives:

```
>> 19 : num
```

3.2.1 Exercise

Evaluate the following expressions by hand and check the answer with Hope.

1.
```
(if (if 1<2 then true else false) or not(2<3)
then 1
else 2) +17;
```

2.
```
if
  (if true then 1 else 2) +17 < 6
then (if 1=3 then 7 else 25) + (if 1<3 then 7 else 25)
else 6;
```
   Mae up some more!

Chapter 4

Functions

4.1 Simple Functions

Functions are the way we can define our own operators. To do this we must make up a name for our function, give its type and then define what it does. For example:

```
f:num -> num;
f(n) <= n*(n+1);
```

This defines a function called `f` of type `num -> num` This means it takes a number and returns a number. If we give our function `f` the value `n` it will give us back `n*(n+1)`

Try inputting

```
f(3);
```

Hope replies:

```
>> 12 :num
```

Because `f(3)` → `3*(3+1)` → `3*4` → `12`.

4.1.1 Exercise

Evaluate the following expressions by hand and check the answer with Hope.

1. `f(f(3));`
2. `if (f(4)<10) then f(2) else f(5);`
3. Define a new function `double` of type `num->num` which returns double its parameter.
4. Define a new function `isOdd` of type `num->bool` which returns true if its parameter is odd and false otherwise. Hint: use the `mod` operator.

4.2 Functions with More than One Parameter

The function `f` above has one parameter. Now consider the following function:

```
implies:bool # bool  -> bool;
implies(a,b) <= not(a) or b;
```

`implies` has two parameters both boolean and it returns a boolean.

4.2.1 Exercise

Evaluate the following expressions by hand and check the answer with Hope.

1. `implies(true,false);`
2. Make a truth table for `implies`

```
max:num # num  -> num;
max(x,y) <= if x>y
            then x
            else y;
```

4.2.2 Exercise

Evaluate the following expressions by hand and check the answer with Hope.

1. `max(3,4);`
2. `max(max(3,5),4);`

4.3 Defining Functions in Terms of Other Functions

We could define a function `isEven:num -> boolean` as follows:

```
isEven:num -> bool;
isEven(n) <=  if n mod 2 = 0
              then true
              else false;
```

or even more simply as

```
isEven:num -> bool;
isEven(n) <= n mod 2 = 0;
```

or we could use the previously defined `isOdd` function:

```
isEven:num -> bool;
isEven(n) <= not (isOdd(n));
```

4.3.1 Exercise

1. Define `maxOf3:num # num # num -> num` in terms of `max` above.
2. Define `maxOf4:num # num # num # num -> num` in terms of `max` and `maxOf3`.
3. Boolean a is `equivalent` to b means a `implies` b and b `implies` a. Define `equivalent` in Hope and make a truth table for it.
4. Write a function `xor:bool # bool -> bool`, standing for 'exclusive or'. If you do not know the meaning of 'exclusive or', search the web for it!

Chapter 5

Recursion

5.1 Defining Recursion

Recursion is when the name of a function occurs on the right hand side of its definition. It is how we achieve repetition.

5.1.1 Examples

```
f:num->num;
f(n) <= if n=0
        then 0
        else n+f(n-1);
```

Notice that we are defining a function called `f` and `f` occurs in the definition of `f`. How do we evaluate `f(0)`? If we substitute n by 0 in the above we get:

```
f(0) -> if 0=0   -> 0 (because 0=0 is true)
        then 0
        else 0+f(0-1)
```

So `f(0)=0`.
What about `f(1)`? To work it out we substitute n by 1 in the equation above to get:

```
f(1) -> if 1=0   ->    1+ f(1-1) (because 1=0 is false) ->  1+ f(0) -> 1+0 -> 1.
        then 0
        else 1+f(1-1)
```

i.e `f(1)=1`
So in order to work out `f(1)` we used that fact that we had previously worked out that `f(0)=0`.

So now let's evaluate `f(2)`: To work it out we substitute n by 2 in the equation above to get:

```
f(1) -> if 2=0   ->    2+ f(2-1) (because 2=0 is false) ->  2+ f(1) -> 2+1 -> 3.
        then 0
        else 2+f(2-1)
```

i.e `f(2)=3`
So in order to work out `f(2)` we used that fact that we had previously worked out that `f(1)=1`.

5.1.2 Exercise

1. Show that `f(3)=6`
2. Show that `f(4)=10`
3. Show that `f(5)=15`
4. What do you think happens if we enter `f(-1)`?
5. Use `f` to do exercise 2.1.1 number 1 again. Notice that `f(n) = 1+2+...+n`.
6. Use `f` to calculate 1+2+...+100.
7. Here is an incorrect function to calculate 1*2*...*n. Run it to see why it is wrong and then try to correct it.

   ```
   factorial:num->num;
   factorial(n) <= if n=0
                   then 0
                   else n*factorial(n-1);
   ```

8. Here is a function to multiply two numbers together. Run it.

   ```
   mult:num # num ->num;
   mult(m,n) <=    if n=0
                   then 0
                   else m+mult(m,n-1);
   ```

   Show that `mult(4,3)=12` and `mult(6,5)=30`

9. Consider the following definition of raising x to the power n:
   $x^0 = 1$ and
   $x^n = x * x^{n-1}$ if $n > 0$.

   Define a function

   ```
   power:num # num ->num;
   ```

   Such that `power(`x`,`n`)` $= x^n$.

5.2 Euclid's Algorithm

The greatest common divisor `gcd` of two numbers is the largest number that divides exactly into both of them. e.g. `gcd(3,4)=1` and `gcd(12,8)=4`. Read `http://en.wikipedia.org/wiki/Euclidean_algorithm`.

5.2.1 Exercise

1. Define

   ```
   gcd:num # num ->num;
   ```

   in Hope.

2. Write a function

   ```
   sumOdds:num->num;
   ```

   Such that `sumodds(n)` is the sum of the first `n` odd numbers.

5.3 Pattern Matching

There is an alternative way to define functions on the integers using pattern matching. For example, the function `f` in Section 5.1 on page 21 could have been defined as follows:

```
f:num->num;
f(0) <= 0;
f(n+1) <= (n+1)+f(n);
```

We now have two rules for `f`. One for when `n=0` and the other for when `n>0`. To evaluate `f(1)` we use the second rule. `f(1)` matches the left hand side of the second rule with `n=0` so to evaluate `f(1)` we evaluate the right hand side of the second rule with `n=0`. This gives `(0+1)+f(0)` which is 1+0 which gives 1.

Similarly, to evaluate `f(2)` we use the second rule. `f(2)` matches the left hand side of the second rule with `n=1` so to evaluate `f(2)` we evaluate the right hand side of the second rule with `n=1`. This gives `(1+1)+f(1)` which is 2+1 which gives 3. etc.

5.3.1 Exercise

Define `factorial`, `mult` and `power` using pattern matching.

Chapter 6

Lists and Tuples

Lists are a kind of *Data Structure*. Data Structures are very important in computer science. Read `http://en.wikipedia.org/wiki/Data_structure`.

6.1 Examples of Lists

Type `[1,2,1];` into Hope. Hope replies with

```
>> [1, 2, 1] : list num
```

Now try: `[true,false,false,false];` Hope replies with

```
>> [true, false, false, false] : list bool
```

6.1.1 Exercises

Work out the value and type of each of the following expressions. Use Hope to check your answers.

1. `[1+1,2,3+1];`
2. `[1<2];`
3. `[1<2,13<5];`
4. `[factorial(5),max(3,7)];`
5. `[if x=1 then 1 else 2, 31, 7-2, power(7,3), 65];`

6.2 Lists of Lists

We can make lists whose elements are themselves list: For example:

```
[[1,2],[3,4,5],[1,2]];
```

is a list of lists of numbers If we type it into Hope, Hope responds with:

```
>> [[1, 2], [3, 4, 5], [1, 2]] : list (list num)
```

6.2.1 Exercises

Work out the type of each of the following expressions. Use Hope to check your answers.

1. `[[true],[false,true]]`
2. `[[[1,2,3]],[[1],[1,1]]]`
3. `[[[[false]]]]`

6.2.2 Errors

Each element of a list must be of the same type. Type

```
[1,false]
```

into Hope. The result is an error message

```
[1, false]
  (::) : alpha # list alpha -> list alpha
  1 : num
  [false] : list bool
type error - argument has wrong type
```

6.2.3 Exercises

Explain why the following list expressions are wrong:

1. `[[1,2,3],1]`
2. `[[1,2],[3,4,5],[true]]`
3. `[[[1,2,3],[4,5]],[6,7,8]]`

6.3 The Empty List, Cons, Head and Tail

There is a special list called the Empty List. Type `[];` into Hope. Hope responds with

```
>> nil : list alpha
```

`nil` is another word for the empty list. Its type is `list alpha`. Here `alpha` means *any* type. It could be a list of integers or a list of bools or a list of anything else.

There is a special operator on lists called 'cons' written `::`. Try

```
>: 1::[2];
```

Hope responds with

```
>> [1, 2] : list num
```

Now try

```
>: 1::[1,1,1,1];
```

Hope responds with

```
>> [1, 1, 1, 1, 1] : list num
```

and
Now try

```
>: true::[false,true];
```

Hope responds with

```
>> [true, false, true] : list bool;
```

So cons takes an element and list and makes a new list who head is the element and who tail is the list. i.e.

```
head: list alpha -> alpha;
head (x::m) <= x;


tail: list alpha -> list alpha;
tail (x::m) <= m;
```

6.3.1 Exercises

Type the above two functions `head` and `tail` into Hope. Evaluate

1. `head [1,2,3,4];`
2. `tail [1,2,3,4];`
3. `head (tail [1,2,3,4]);`
4. `tail (tail [1,2,3,4]);`
5. `head(tail (tail [1,2,3,4]));`
6. `tail(tail (tail [1,2,3,4]));`
7. `head(tail(tail (tail [1,2,3,4])));`
8. `tail(tail(tail (tail [1,2,3,4])));`
9. `head(tail(tail(tail (tail [1,2,3,4]))));`
   This one gives an error. Explain why.

6.4 Defining Functions on Lists

Here is a function for adding a list of numbers together:

```
sum: list(num) -> num;
sum [ ] <= 0;
sum(x::m) <= x + sum(m);
```

We read this as follows: The sum of the empty list of integers is zero. To sum the list of integers whose head is `x` and whose tail is `m`, we sum the tail, `m` and add the head `x` to the result.

How is this evaluated? Suppose we ask Hope to evaluate `sum[2]`. Now `[2]` is the list whose head is 2 and whose tail is `[]`. So `[2]=2::[]`. So `sum[2]=sum(2::[])`. This matches the second rule for sum with `x=2` and `m=[]`.

So we rewrite the right hand side of the second rule with `x=2` and `m=[]` to give `2+sum([])`;

By the first rule for `sum`, `sum([])=0` so the result of `sum[2]` is `2+0` which is 2.

6.4.1 Exercises

1. Using the fact that `sum[2]=2` Show using the same approach as above that `sum[3,2]=5`

2. Using the fact that `sum[3,2]=5`. Show using the same approach as above that `sum[4,3,2]=9`.

3. Write a function `multAll` which multiplies a list of numbers together.

6.4.2 Counting the number of Elements in a List

The empty list has zero elements. The list `x::m` has one more element than the list `m`. So:

```
length: list(alpha) -> num;
length [ ] <= 0;
length (x::m) <= x + length(m);
```

Notice `length` takes a `list (alpha)`, i.e. a list of any type. Since counting the elements of a list of any type is the same.

6.4.3 Exercises

Evaluate

1. `length [1,2,3,4];`

2. `length(tail [1,2,3,4]);`

3. `length [[1,2,3],[4]];`

   Explain this one.

4. `length(tail (tail [1,2,3,4]));`

5. `length [[[2,3],[4,5]]];`

   Explain this one. Make a list of the same type of length 2.

6. Explain what this function does:

   ```
   downTo: num -> list(num);
   downTo 0 <= [];
   downTo (n+1) <= (n+1):: downTo(n);
   ```

7. Explain what this function does:

   ```
   f: num -> num;
   f(n) <= multAll(downTo(n));
   ```

   Have you seen a function that does this already?

8. (Very Difficult) Try and define a function

   ```
   upTo: num -> list(num);
   ```

   Such that

   ```
   upTo(n)=[1,2,...,n];
   ```

6.4.4 Appending Two Lists

We want to be able to append (concatenate) one list on to the end of another. To do this we define the `append` function.

```
append: list alpha # list alpha -> list alpha
```

The `append` function takes two lists as parameters and returns the list consisting of the second list 'stuck on' the end of the first list. As you might now expect, `append` is defined recursively. There are two rules for the `append` function. The first rule is when the left hand list is empty in which case we simply return the second list. i.e.

```
append ([],k) <= k;
```

The second rule is for when the left hand list is not empty. In which case, we recursively `append` the tail of the left list to the right list and then `cons` the head of the left list onto the result:

```
append (x::m,k) <= x::append(m,k);
```

6.4.5 Exercises

Define `append` in Hope and try the following:

1. `append([],[]);`
2. `append([1,2,3],[]);`
3. `append([true,true,false],[false]);`
4. `append([1,2,3],1::[]);`
5. `append([[1,2,3]],[1::[]]);`
6. `append([1,2,3],[true,false true]);`

   This one gives an error. Why?

7. If you didn't do it before, now define the `upTo` function mentioned earlier. To make `upTo(n+1)`, we make `upTo(n)` and then `append` the singleton list `[n+1]` to its right.

6.4.6 Reversing a List

To reverse a non empty list we reverse its tail and then append a singleton list consisting of just its head to the right.

```
reverse: list(alpha) -> list(alpha);
reverse [] <= [];
reverse (x::m) <= append(reverse(m),[x]);
```

6.4.7 Exercises

1. Write a function

   ```
   occurrences: alpha # list(alpha) -> num;
   ```

   Such that `occurrences(x,m)` returns the number of occurrences of `x` in the list `m`. For example `occurrences(3,[3,4,5,3])` is 2. and `occurrences(7,[3,4,5,3])` is 0.

2. Write a function `maxList` which returns the largest number in a list of non-negative integers. Hint: Define the maximum of the empty list to be zero and use the `max:num # num -> num` function you defined earlier on the course.

3. Write a function `allSame: num # alpha -> list(alpha)` such that `allSame(n,y)` returns a list of `n` `y`s. For example `allSame(3,true)` would give `[true,true,true]`.

6.5 Tuples

In Hope, lists have to have elements all of the same type but if we want data structures where the elements can be of different type then we can use tuples. Mathematically speaking, tuples are elements form the *Cartesian Product* of Sets.

6.5.1 Examples

Type the following into Hope and observe the type.

1. `>: (1,3);`
   `>> (1, 3) : num # num`
2. `>: (1,true,false);`
   `>> (1, true, false) : num # bool # bool`
3. `>: ([1],true,56);`
   `>> ([1], true, 56) : list num # bool # num`
4. `>: ([1,2],[true],45,1,false);`
   `>> ([1, 2], [true], 45, 1, false) : list num # list bool # num # num # bool`
5. `>: ((1,2),[true],45,(1,false));`
   `>> ((1, 2), [true], 45, 1, false) : (num # num) # list bool # num # num # bool`

6.5.2 Exercises

Make up some expressions of the following types:

1. `list num # bool # num`
2. `list(num # num # bool # bool)`
3. `list num # list bool # num`
4. `list(list bool # num)`

Chapter 7

Problem Solving using Stepwise Refinement

When you have a big problem to solve you have to break it down into smaller problems that you may have not yet solved. You repeat the process until everything is solved and then you put it all together to get your solution. This is called *stepwise refinement*. Another word for this process is *top-down design*.

7.1 Example

7.1.1 Specifying the Problem

Problem: write a function `split` which takes a list and divides it into two lists of equal length -'split' down the middle. For example `split([1,2,3,4]` should give `([1,2],[3,4])`;

Now there is a problem with this 'specification': What should our function do if the list has an odd length? The first thing we need to do when solving a problem is to make sure that the problem is well-defined. Clearly here it is not. So let's have another go:

Problem: write a function `split` which takes a list and divides it into two lists of equal length if the list's length is even. if the list's length is odd then the right hand list should have the extra element. For example `split([1,2,3,4]` should give `([1,2],[3,4])` and `split([1,2,3,4,5]` should give `([1,2],[3,4,5])`
That's better! Now let's try and solve the problem using top-down design.

7.1.2 Solving the Problem

First we need the type of our function `split`:

```
split: list alpha -> list alpha # list alpha;
```

What does split do?

```
split(k) <= (firstHalf k,secondHalf k);
```

This will work fine provided that we have functions `firstHalf` and functions `secondHalf` with the following specification:

Specification: `firstHalf` is a function which takes a list `k` and returns a list consisting of the first `length(k) div 2` elements of the list. For example `firstHalf [1,2,3,4,5]` should give `[1,2]` since the length is 5 and 5 div 2 is 2.

Now what about `secondHalf`? `secondHalf` is a function which takes a list `k` and returns a list consisting of k after the first `length(k) div 2` elements k have been thrown away. So we have

```
firstHalf: list(alpha) -> list(alpha);
firstHalf k <= keep( (length k) div 2 ,k);

secondHalf: list(alpha) -> list(alpha);
secondHalf k <= throwAway( (length k) div 2 ,k);
```

So all that is left to do is specify and design `keep` and `throwAway`.

The function `keep` takes a number `n` and a list `k` and returns a list consisting of the first `n` elements of `k`. That's easy:

```
keep: num # list(alpha) -> list(alpha);
keep(0,k) <= [];
keep (n+1,x::k) <= x::keep(n,k);
```

The function `throwAway` takes a number `n` and a list `k` and returns list `k` with the first `n` elements thrown away.

```
throwAway: num # list(alpha) -> list(alpha);
throwAway(0,k) <= k;
throwAway (n+1,x::k) <= throwAway(n,k);
```

Our task is now complete. This gives is the complete Hope program:

```
throwAway: num # list(alpha) -> list(alpha);
throwAway(0,k) <= k;
throwAway (n+1,x::k) <= throwAway(n,k);

keep: num # list(alpha) -> list(alpha);
keep(0,k) <= [];
keep (n+1,x::k) <= x::keep(n,k);

length: list(alpha) -> num;
```

```
length [] <= 0;
length (x::k) <= 1 + length k;

firstHalf: list(alpha) -> list(alpha);
firstHalf k <= keep( (length k) div 2 ,k);

secondHalf: list(alpha) -> list(alpha);
secondHalf k <= throwAway( (length k) div 2 ,k);

split: list alpha -> list alpha # list alpha;
split(k) <= (firstHalf k,secondHalf k);
```

7.2 Another Example - Sorting a List of Integers

The problem is to define a function `sort` which take a list of numbers and returns a list of the same numbers but sorted in ascending order. For example `sort [1,6,2,4,0,2,2,3]` would give `[0,1,2,2,2,3,4,6]`. Again this problem is not fully specified. What about the empty list? It makes sense to define `sort [ ] = [ ]`. So now we are ready to tackle the problem. We can try:

```
sort: list num -> list num;
sort [ ] <= [ ]
sort (x::m) <= ?
```

Again we must think recursively: The right hand side will probably have the expression `sort (m)`. In other words the sorted tail of the list. The trick is to imagine that this expression `sort (m)` has correctly sorted the tail of list. So what do we have to do with the head `x` so that the whole list `x::m` is correctly sorted? We must *insert* it into the correct place in the sorted tail `sort m`. This gives us:

```
sort: list num -> list num;
sort [ ] <= [ ];
sort (x::m) <= insert(x,sort m);
```

So now we have reduced the sorting problem to a simpler inserting problem. How do we insert a number into the correct place in a already sorted list of numbers? Well form the expression `insert(x,sort m)` we can see the type of `insert` is `num # list(num)-> list(num)`; So let's try:

```
insert: num # list num -> list num;
insert(y, []) <= [y];
insert(y, x::k) <= ?
```

If the list we are inserting into is empty then it's easy. We create a singleton list consisting of the number we are inserting. If the list we are inserting in is not empty then what do we do?

Don't forget that the list we are inserting into is sorted. So what we need to do is compare `y` with the head of the list `x`. If `y <x` then we must put `y` at the beginning of the result:

```
insert: num # list num -> list num;
insert(y, []) <= [y];
insert(y, x::k) <= if y <x
                   then y::(x::k)
                   else ?;
```

Well, if `y` is not less than `x` then we want to keep `x` as the head of the result. The tail of the result will be the achieved by inserting `y` into the tail `k`. To give:

```
insert: num # list num -> list num;
insert(y, []) <= [y];
insert(y, x::k) <= if y <x
                   then y::(x::k)
                   else x::insert(y,k);
```

So we now have our complete sorting program:

```
insert: num # list num -> list num;
insert(y, []) <= [y];
insert(y, x::k) <= if y <x
                   then y::(x::k)
                   else x::insert(y,k);

sort: list num -> list num;
sort [ ] <= [ ];
sort (x::m) <= insert(x,sort m);
```

Try it out!

Chapter 8

Assignment

1. Without using `if`, write a function

   ```
   isDivBy3: num -> bool;
   ```

   which returns true if and only if its argument is divisible by 3.

2. Without using `if`, write a function

   ```
   isDivBy: num # num -> bool;
   ```

   which returns true if and only if its first argument is divisible by its second argument.

3. Re-implement `isDivBy3` using `isDivBy`.

4. There is a boolean operator called NAND. Search for it on the Web and implement it in Hope.

5. Write a function `contains`

   ```
   contains: alpha # list(alpha) -> bool;
   ```

   `contains(x,k)` returns true if and only if list `k` contains `x`.

6. A list that is the same forwards and backwards is called a *palindrome*. Specify, design and write a function

   ```
   isPalindrome: list alpha -> bool;
   ```

   which returns true if and only if its argument is a palindrome. e.g. `isPalindrome [1,2,1]` is true but `isPalindrome [1,2]` is false.

7. Specify, design and write a function which returns the middle element of a list. You decide what the middle element of a list of even length.

8. Specify, design and write a function `remove`,

   ```
   remove: alpha # list alpha -> list(alpha);
   ```

   Such that `remove(x,k)` removes all occurrences of `x` from the list `k`. For example `remove(2,[3,2,3,2,3,4])` gives `[3,3,3,4]`.

9. Specify, design and write a function `removeDuplicates`,

   ```
   removeDuplicates:  list (alpha) -> list(alpha);
   ```

   Such that `removeDuplicates k` removes all duplicates from k. For example `removeDuplicates([3,2,3,2,3,4])` gives `[3,2,4]`.

10. Specify and Use Stepwise refinement to design and write a function

    ```
    isPermutation: list alpha # list alpha -> bool;
    ```

    `isPermutation(k,m)` returns true if and only if `k` and `m` are permutation of each other. For example `[1,2,3,1]` and `[1,3,1,2]` are permutations but `[1,2,3,1]` and `[1,3,2]` are not.