CONCEPTS OF PROGRAMMING LANGUAGE

RECURSION, LOOPS, AND INVARIANT PROGRAMMING

Recursion and loops

- In the previous lesson we saw SumDigitsR:

```
fun {SumDigitsR N}
    if (N==0) then 0
    else (N mod 10) + {SumDigitsR (N div 10)}
    end
end
```

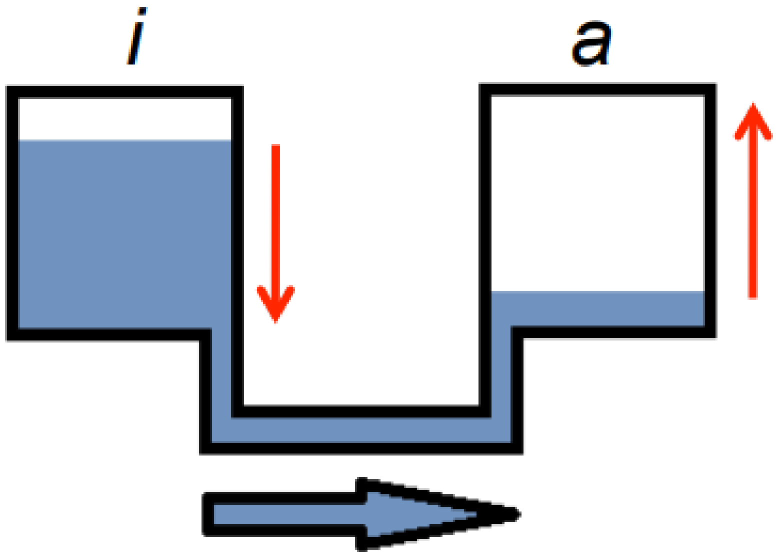
- The recursive call and the condition together act **like a loop**: a calculation that is repeated to achieve a result
  - Each execution of the **function body** is one iteration of the loop
- Recursion can be used to make a loop
  - In this lesson we will go to the root of this intuition

Invariant programming

- A loop is a part of a program that is repeated until a condition is satisfied
  - Loops are an important technique in all paradigms
  - Loops are a special case of recursion, called **tail recursion,** where the recursive call is the last operation done in the function body
- We will give a general technique, **invariant programming**, to program correct and efficient loops
  - Loops are often very difficult to get exactly right, and invariant programming is an excellent way to achieve this
- This applies to both **declarative** and **imperative** paradigms
- New concepts introduced in this lesson
- Accumulator, principle of communicating vases

Principle of communicating vases

- We invent a formula that splits the work into two parts:
  - n! = i! * a
- We start with i=n and a=1
- We decrease i and increase a, keeping the formula true
- When i=0 then a is the result
- Here’s an example when n=4:
  - 4! = 4! * 1
  - 4! = 3! * 4
  - 4! = 2! * 12
  - 4! = 1! * 24
  - 4! = 0! * 24



Question

- The name "communicating vases" comes from the fact that:
  - We can decrease a variable and in the same time increase another, just like pouring water from one vase to another.

Exercise:
Sum of Digit with invariant programming

Sum of digits using invariant programming

- Each recursive call handles one digit
- So we divide the initial number n into its digits:
  - $n = (d_{k-1}d_{k-2}\cdots d_2d_1d_0)$ (where d_i is a digit)
- Let's call the sum of digits function s(n)
- Then we can split the work in two parts:
  - $s(n) = \underbrace{s(d_{k-1}d_{k-2}\cdots d_i)}_{S_i} + \underbrace{(d_{i-1} + d_{i-2} + \cdots + d_0)}_{A}$
- s_i is the work still to do and a is the work already done
- To keep the formula true, we set i' = i+1 and a' = a+di
- When i=k then s_k=s(0)=0 and therefore a is the answer

Example execution

- Example with n=314159:

s(n) = s(dk-1dk-2⋯ di) + (di-1 + di-2 + ⋯ + d0)

  - s(314159) = s(314159) + 0
  - s(314159) = s(31415) + 9
  - s(314159) = s(3141) + 14
  - s(314159) = s(314) + 15
  - s(314159) = s(31) + 19
  - s(314159) = s(3) + 20
  - s(314159) = s(0) + 23 = 0 + 23 = 23

Final program

- $S = (d_{k-1}d_{k-2}\cdots\ d_i\ )$
- $A = (d_{i-1} + d_{i-2} + \cdots + d_0\ )$

```
fun {SumDigits2 S A}
    if S==0 then A
    else
        {SumDigits2 (S div 10) A+(S mod 10)}
    end
end
```

What can we learn from these examples?

- We have now seen two examples of recursive functions
  - Factorial
  - Sum of digits
- For each example we have seen two versions
  - A version based on a simple mathematical definition
  - A version designed with invariant programming
- The second version has two interesting properties
  - It has two arguments; one of the two is an accumulator
  - The recursive call is the last operation in the function body (tail recursion)

The importance of tail recursion

- Let us now take a closer look at why tail recursion is important
- We will do a detailed comparison of the execution of Fact1 and Fact2
- We will see why Fact2 (with tail recursion) is more efficient than Fact1 (no tail recursion)
  - Fact1 is based on a simple mathematical definition
  - Fact2 is designed with invariant programming

Comparing Fact1 and Fact2

- Tail recursion is when the recursive call is the last operation in the function body
- N * {Fact1 N-1} % No tail recursion
  - (*)After Fact1 is done, we must come back for the multiply. Where is the multiplication stored? On a stack!
  - The function is not tail recursive because the value returned by fact(n-1) is used in fact(n) and call to fact(n-1) is not the last thing done by fact(n)

- {Fact2 I-1 I*A} % Tail recursion
  - The recursive call does not come back!
  - All calculations are done before Fact2 is called.

    No stack is needed (memory usage is constant).

Comparing functional and imperative loops

- A while loop in the functional paradigm:

```
fun {While S}
   if {IsDone S} then S
   else {While {Transform S}} end /* tail recursion */
end
```

- A while loop in the imperative paradigm:
  (in languages with multiple assignment like Java and C++)

```
state whileLoop(state s) {
   while (!isDone(s))
      s=transform(s); /* assignment */
   return s;
}
```

- In *both* cases, invariant programming is an important design tool

Summary and a bigger example

- We summarize this lesson in a few sentences
  - A recursive function is equivalent to a loop if it is tail recursive
  - To write functions in this way, we need to find an accumulator
  - We find the accumulator starting from an invariant using the principle of communicating vases
  - This is called invariant programming and it is the only reasonable way to program loops
  - Invariant programming is useful in all programming paradigms
- Now let’s tackle a bigger example!

A bigger example: calculating X^N

- Let's use invariant programming to define a function {Pow X N} that calculates X^N ($N \geq 0$)
- Let's start with a naive definition of x^n:

  $x^0 = 1$

  $x^n = x * x^{n-1}$ when $n>0$

- This gives a first program for {Pow X N} :

```
fun {Pow1 X N}
    if N==0 then 1
    else X*{Pow1 X N-1} end
end
```

- This function is highly inefficient in both time and space! Why?

- Using a better definition of X^N

```
declare
fun {Powerx N I A}
   if I==0 then A
   else
      {Powerx N I-1 A*N }
   end
end

{Browse {Powerx 3 3 1}}
```

Invariants and goals

- Changing one part of the invariant forces the rest to change as well, because the invariant must remain true
  - The invariant’s truth drives the program forward
- Programming a loop means finding a good invariant
  - Once a good invariant is found, coding is easy
- Learn to think in terms of invariants!
- Using invariants is a form of goal-oriented programming
  - We will see another example of goal-oriented programming when we program with Lecture trees

LISTS AND PATTERN MATCHING

List – Coding

```
declare
L=[1 2 3]
{Browse L}
M={Append L L}
{Browse M}
declare
N=nil
{Browse N}
{Browse {Append L N]}
declare
F=[1+1 1.2 salwa]
{Browse F}
```

We are in the Function paradigm which mean there isn't loop and values are immutable

Lists can contain lists and elements of different type.

Definition of a list

- A list is a recursive data type: we define it in terms of itself
  - Recursion is used both for **computations and data**!
  - We need to know this when we **write functions on lists**
- A list is either an empty list or a pair of an element followed by another list
  - We will build large list from small list
  - This definition is recursive because it defines lists in terms of lists. There is **no infinite** regress because the definition is used constructively to build larger lists from smaller lists.
- Let’s introduce a formal notation

Syntax definition of a list

- Using an EBNF grammar rule we write:
- <List T> ::= nil | T ‘|’ <List T>
  - ::= is defined as
- This defines the textual representation of a list
- EBNF = Extended Backus-Naur Form
  - Invented by John Backus and Peter Naur
  - <List T> represents a list of elements of type T
  - T represents one element of type T
- Be careful to distinguish between | and ‘|’ : the first is part of the grammar notation (it means “or”), and the second is part of the syntax being defined

Some examples of lists

- According to the definition (if T is integers):

  nil

  10 | nil

  10 | 11 | nil

  10 | 11 | 12 | nil
- What about the bracket notation we saw before?
  - It is not part of the recursive definition of lists; it is an extra called syntactic sugar

Type notation

- <Int> represents an integer; more precisely, it is the set of all syntactic representations of integers
- <List <Int>> represents the set of all syntactic representations of lists of integers
- T represents the set of all syntactic representations of values of type T; we say that T is a type variable
  - Do not confuse a type variable with an identifier or a variable in memory! Type variables exist only in grammar rules.

Representations for lists

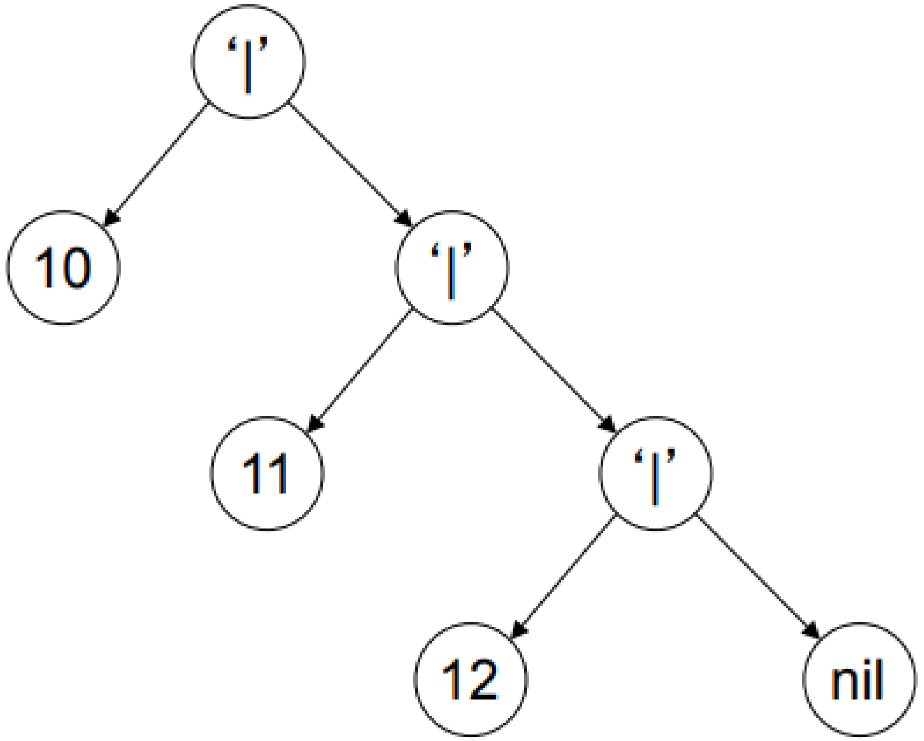
- The EBNF rule gives one textual representation
  - <List <Int>> ⇒
    10 | <List <Int>> ⇒
    10 | 11 | <List <Int>> ⇒
    10 | 11 | 12 | <List <Int>> ⇒
    10 | 11 | 12 | nil

We repeatedly replace the left-hand side of the rule by a possible value, until no more can be replaced

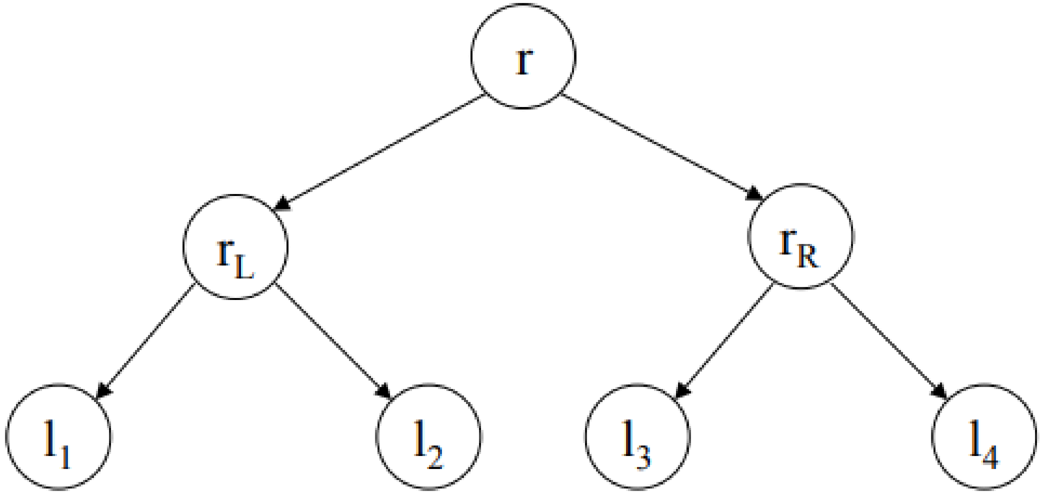
- Oz allows another textual representation
  - Bracket notation: [10 11 12]
  - In memory, [10 11 12] is identical to 10 | 11 | 12 | nil
  - Different textual representations of the same thing are called syntactic sugar

Graphical representation of a list

- Graphical representations are very useful for reasoning
  - Humans have very powerful visual reasoning abilities
- We start from the leftmost pair, namely 10 | <List <Int>>
  - We draw three nodes with arrows between them
  - We then replace the node <List <Int>> as before
- This is an example of a more general structure called a tree



Trees and binary trees



- A **tree** is either a leaf node (which is an empty tree) or a root node with arrows to a set of trees (called subtrees)
- A **binary tree** is a tree where all root nodes have exactly two subtrees (usually called left and right)

List is a special Case of Tree

Tail recursion for lists

```
declare X1 X2 in
X1=6|X2
{Browse X1}
```

```
declare X3 in
X2=7|X3
X3=nil
{Browse X1}
```

```
%Build-in function
%Guess what if you want to display 7 as a
head
{Browse X1.2.1}
{Browse X1.2.2}
```

Tail recursion for lists

declare X1 X2 in
X1=6|X2
{Browse X1}
6|_
Underscore is Unbound variable

declare X3 in
X2=7|X3
{Browse X1}
X3=nil

{Browse X1}

6|7|_

- in the same single statement in browser
- So the browser can update the binding of single-assignment variables.

[6 7]

- So the browser knows that it's actually a complete list.
- It is with bracit notation but are the same
- {Browse [6 7] ==6|7|nil} TRUE
- Remember the bracket notation is a syntactic sugar.

MORE FUNCTIONS ON LISTS

Next Lecture

THANK
S