Lesson 28

• Control Flow
  – Recursion
  – Continuations
Recursion

• Recursion: subroutines that call themselves directly or indirectly (mutual recursion)
• Typically used to solve a problem that is defined in terms of simpler versions, for example:
  – To compute the length of a list, remove the first element, calculate the length of the remaining list in \( n \), and return \( n+1 \)
  – Termination condition: if the list is empty, return 0
• Iteration and recursion are equally powerful in theoretical sense
  – Iteration can be expressed by recursion and vice versa
• Recursion is more elegant to use to solve a problem that is naturally recursively defined, such as a tree traversal algorithm
• Recursion can be less efficient, but most compilers for functional languages are often able to replace it with iterations
Tail-Recursive Functions

• *Tail-recursive functions* are functions in which no operations follow the recursive call(s) in the function, thus the function returns immediately after the recursive call:

  - tail-recursive
    ```
    int trfun() {
        ... 
        return trfun();
    }
    ```
  
  - not tail-recursive
    ```
    int rfun() {
        ... 
        return 1+rfun();
    }
    ```

• A tail-recursive call could *reuse* the subroutine's frame on the runtime stack, since the current subroutine state is no longer needed
  – Simply eliminating the push (and pop) of the next frame will do

• In addition, we can do more for *tail-recursion optimization*: the compiler replaces tail-recursive calls by jumps to the beginning of the function
Tail-Recursion Optimization

• Consider the GCD function:
  ```c
  int gcd(int a, int b)
  { if (a==b) return a;
    else if (a>b) return gcd(a-b, b);
    else return gcd(a, b-a);
  }
  ```
  a good compiler will optimize the function into:
  ```c
  int gcd(int a, int b)
  { start:
    if (a==b) return a;
    else if (a>b) { a = a-b; goto start; }
    else { b = b-a; goto start; }
  }
  ```
  which is just as efficient as the iterative version:
  ```c
  int gcd(int a, int b)
  { while (a!=b)
    { if (a>b) a = a-b;
      else b = b-a;
      return a;
    }
  ```
Converting Recursive Functions to Tail-Recursive Functions

- Remove the work after the recursive call and include it in some other form as a computation that is passed to the recursive call.
- For example, the non-tail-recursive function computing $\sum_{n=\text{low}}^{\text{high}} f(n)$

\[
\text{summation} = (f, \text{low}, \text{high}) -> \\
\begin{align*}
\text{if} (\text{low} == \text{high}) & \text{ then } (f \text{ low}) \\
\text{else } & (f \text{ low}) + \text{summation} (f, \text{low + 1}, \text{high})
\end{align*}
\]

can be rewritten into a tail-recursive function:

\[
\text{summationTR} = (f, \text{low}, \text{high}, \text{subtotal}) -> \\
\begin{align*}
\text{if} (\text{low} == \text{high}) & \text{ then } \text{subtotal} + (f \text{ low}) \\
\text{else } & \text{summationTR} (f, \text{low + 1}, \text{high}, \text{subtotal} + (f \text{ low}))
\end{align*}
\]
Converting recursion into tail recursion: Example

• Here is the same example in C:

```c
typedef int (*int_func)(int);

int summation(int_func f, int low, int high) {
    if (low == high)
        return f(low)
    else
        return f(low) + summation(f, low+1, high);
}
```

• rewritten into the tail-recursive form:

```c
int summationTR(int_func f, int low, int high, int subtotal) {
    if (low == high)
        return subtotal+f(low)
    else
        return summationTR(f, low+1, high, subtotal+f(low));
}
```
When Recursion is Bad

- The Fibonacci function implemented as a recursive function is very inefficient as it takes exponential time to compute:

\[
\text{fib} = \begin{cases} 
\text{if } n == 0 \text{ then } 1 \\
\text{else if } n == 1 \text{ then } 1 \\
\text{else fib}(n - 1) + \text{fib}(n - 2) 
\end{cases}
\]

- with a tail-recursive helper function, we can run it in O(n) time:

\[
\text{fibTR} = \begin{cases} 
\text{let fibhelper}(f1, f2, i) = 
\text{if } (n == i) \text{ then } f2 \\
\text{else fibhelper}(f2, f1 + f2, i + 1) 
\end{cases}
in \text{fibhelper}(0, 1, 0)
\]
Continuation-passing Style

• Makes **control** explicit in functional programming (including evaluation order of operands/arguments, returning from a function, etc.)

• A **continuation** is a function representing “the rest of the program” taking as argument the current result

• Functions have an additional (last) argument, which is a continuation

• Primitive functions have to be encapsulated in CPS ones

Encapsulation of primitive operators

\[ (*&) \ x \ y \ k = k (x * y) \]

\[ (+&) \ x \ y \ k = k (x + y) \]

\[ (==&) \ x \ y \ k = k (x == y) \]

\[ \text{sqrtK} \ x \ k = k (\text{sqrt} \ x) \]
Making evaluation order explicit

• Function call arguments must be either variables or lambda expressions (not more complex expressions)

**Direct style:** evaluation order is implicit

\[
\text{diag} \ x \ y = \sqrt{(x \times x) + (y \times y)} \\
\text{diag} 3 4 \rightarrow 5.0
\]

**Continuation-passing style:** evaluation order is explicit

\[
\text{diagK} \ x \ y \ k = \\
\quad (*&) \ x \ x \ (\lambda x2 \rightarrow \\
\quad \quad (*&) \ y \ y \ (\lambda y2 \rightarrow \\
\quad \quad \quad (+&) \ x2 \ y2 \ (\lambda x2y2 \rightarrow \\
\quad \quad \quad \quad (\text{sqrtK} \ x2y2 \ k))) \\
\text{diagK} 3 4 (\lambda x \rightarrow x) \rightarrow 5.0
\]
Non-tail-recursive functions cause continuation in recursive call to grow

**Direct style**: non-tail-recursive factorial

```plaintext
factorial n = if (n == 0) then 1
  else n * factorial (n - 1)
```

**Continuation-passing style**: non-tail-recursive factorial

```plaintext
factorialK n k = (==&) n 0 (\b ->
  if b then (k 1) else
  (-&) n 1 (\nml ->
    factorialK nml (\f-> (((& n f k)))))
```
Tail-recursive functions: continuation in recursive call is identical

**Direct style**: tail-recursive factorial

```haskell
factorialTR n = faux n 1
faux n a = if (n == 0) then a
            else faux (n - 1) (n * a) -- tail recursive
```

**Continuation-passing style**: tail-recursive factorial

```haskell
factorialTRK n k = fauxTR n 1 k

fauxTR n a k = (==&\) n 0 (\b ->
  if b then (k a) else
  (-&\) n 1 (\nm1 ->
    (*&\) n a (\nta ->
      (fauxTR nm1 nta k)))))
```
On continuation-passing style

• If all functions are in CPS, no runtime stack is necessary: all invocations are **tail-calls**
• The continuation can be replaced or modified by a function, implementing almost arbitrary control structures (exceptions, goto’s, …)
• Continuations used in denotational semantics for goto’s and other control structure (eg: bind a label with a continuation in the environment)

**Continuation-passing style**: returning **error** to the top-level

```plaintext
sqrt n k = if (n < 0) 'error
   else k (safe-sqrt n)
```

**Direct style**: the callers should propagate the error along the stack