CAS CS 320
Lec 10

Stream & Python Generator

1. Lazy (review)
    a. Instead of building infinite list, we build mechanism that continues to generate list when called
    b.
    ```
    fun from(x) = fn() => (*fn() means it is lambda → value → do not compute*)
            x :: from(x+1)
    ```
    c.
    ```
    datatype ‘a strmcon = strmcon_nil | strmcon_cons of ‘a * ‘a stream
    (* stream constructor consists of the first value of the stream and the rest *)
    ```
    d. ‘a stream has type (unit -> ‘a strmcon) (*often named as thunk*)
    e.
    ```
    fun from(x) = fn() => (*updated version*)
            strmcon_cons(x, from(x+1))
    ```
2. Stream
    a.
    ```
    fun stream_map(fxs: ‘a stream, fopr: ‘a -> ‘b): ‘b stream = fn() =>
    (*when procedure is called, we generate an element*)
    (*returns immediately, it does not do much work → use fn()*)
            case fxs() of
                    strmcon_nil => strmcon_nil
                    | strmcon_cons(x1, fxs) => strmcon_cons(fopr(x1),
                    stream_map(fxs, fopr))
    ```
    b.
    ```
    fun stream_filter(fxs: ‘a stream, test: ‘a -> bool): ‘a stream = fn() =>
            case fxs() of
            strmcon_nil => strmcon_nil
            | strmcon_cons(x1, fxs) =>
                    if not(test(x1)) then stream_filter(fxs, test)()
                    (* make stream into a stream constructor by making it unit type by
                    putting () *)
                    else strmcon_cons(x1, stream_filter(fxs, test))
    ```
    c.
    ```
    fun sieve(fxs: int stream): int stream = fn() =>
    let
            val strmcon_cons(p1, fxs) = fxs()
    in
            strmcon_cons(p1,sieve(stream_filter(fxs, fn x1 => x1 mod p1 > 0)))
    end
    val thePrimes = sieve(from(2))
    val fxs = thePrimes
    val strmcon_cons(p0, fxs) = fxs() (* value is 2*)
    ```

```
    val strmcon_cons(p1, fxs) = fxs() (*value is 3*)
    val strmcon_cons(p2, fxs) = fxs() (*value is 5*)
d.  fun stream_append(fxs: ‘a stream, fys: ‘a stream): ‘a stream = fn() =>
        case fxs() of
        strmcon_nil => fys()
        | strmcon_cons(x1, fxs) =>
        strmcon_cons(x1, stream_append(fxs,fys))
    (* if fxs is infinite stream, we cannot append two streams*)
    Therefore,
    fun stream_alter(fxs: ‘ a stream, fys: ‘a stream): ‘a stream = fn() =>
        case fxs() of
        strmcon_nil => fys()
        | strmcon_cons(x1, fxs) => strmcon_cons(x1, stream_alter(fys, fxs))
    (*switch fxs and fys back and forth*)
e.  fun stream_zip(fxs: ‘a stream, fys: ‘b stream): (‘a * ‘b) stream = fn() =>
        case fxs() of
        strmcon_nil => strmcon_nil
        | strmcon_cons(x1, fxs) =>
            case fys() of
            strmcon_nil => strmcon_nil
            strmcon_cons(y1, fys) => strmcon_cons((x1, y1), stream_zip(fxs,
            fys))
f.  fun stream_z2map(fxs: ‘a stream, fys: ‘b stream, fopr: (‘a * ‘b) -> ‘c): ‘c stream =
    fn() =>
        stream_map(stream_zip(fxs,fys),fopr)()
g.  fun stream_tabulate(n0: int, fopr: int -> ‘a): ‘a stream =
        let
            fun loop1(i0: int): ‘a stream = fn() =>
                strmcon_cons(fopr(i0), loop1(i0+1))
            fun loop2(i0: int): ‘a stream = fn() =>
                if i0 < n0 then strmcon_cons(fopr(i0), loop2(i0+1))
                else strmcon_nil
        in
        end
```

3. DFS and BFS

```
a.  datatype node = NODE of int
b.  fun node_get_neighbors(node): node list = []
c.  fun dfs_walk(node): node stream = fn() =>
        strmcon_cons(node, dfs_walk_list(node_get_neighbors(node)))
d.  fun dfs_walk_list(nodes): node stream = fn() =>
```

```
case nodes of
nil => strmcon_nil
| (node::nodes) =>
        strmcon_cons(node, dfs_walk_list(node_get_neighbors(node) @
nodes))
```

e.
```
fun bfs_walk(node: int list): node stream = fn() =>
        strmcon_cons(node, bfs_walk_list(node_get_neighbors(node)))
```

f.
```
fun bfs_walk_list(nodes: int list): node stream = fn() =>
        case nodes of
        nil => strmcon_nil
        | (node::nodes) =>
                strmcon_cons(node, bfs_walk_list(nodes @
        node_get_neighbors(node)))
```

4. Python generators

a.
```
def generator_tabulate(n0, fopr):
        if n0 >= 0:
                i0 = 0
                while i0 < n0:
                        yield fopr(i0)
                        i0 = i0 + 1
        else:
                i0 = 0
                while True:
                        yield fopr(i0)
                        i0 = i0 + 1
        return None
```

b.
```
fxs = generator_tabulate(-1, lambda x: x * x)
next(fxs) # gives 0
next(fxs) # gives 1
next(fxs) # gives 4
#System remembers where you stopped and resumes from the place it stopped
```

c.
```
fxs = generator_tabulate(3, lambda x: x * x)
next(fxs) # gives 0
next(fxs) # gives 1
next(fxs) # gives 4
next(fxs) # error occurs, stop iteration
```

d.
```
fxs = generator_tabulate(5, lambda x: x * x)
list(fxs) → gives [0,1,4,9,16] OR set(fxs) OR tuple(fxs)
```

If you call again, it comes out [] since there is no more e.

```
def generator_append(fxs, fys):
    yield from fxs
    yield from fys
    # if there is no more element in fxs, stop iteration and go to fys and same thing
    # easy to use but there is efficiency issue (try to ignore yield from)
```

f.

```
def generator_filter(fxs, test):
    while True:
        x1 = next(fxs)
        if test(x1):
            yield x1
```

g.

```
def generator_sieve(fxs):
    p1 = next(fxs)
    yield p1
    yield from generator_sieve(generator_filter(fxs, lambda x1: x1 % p1 > 0))
```

h.

```
fxs = generator_tabulate(-1, lambda x: x + 2)
fps = generator_sieve(fxs)
next(fps) # gives 2
next(fps) # gives 3
next(fps) # gives 5
next(fps) # gives 7
```

i.