Today’s plan

- Continuations as “rest of computation”
- Continuation-passing style (CPS): A (new) programming discipline
- CPS interpreters
Why continuations?

- Continuations pop up in many fields of programming languages.
- Continuations are unavoidable if you *study* programming languages:
  - (Denotational or monadic) semantics of goto, exceptions, co-routines, generators, backtracking, ...
  - Program analysis: (control-flow analysis)
  - Logics
- Continuations are valuable tools if you *use* programming languages:
  - (Higher-order or functional) implementations of goto, exceptions, co-routines, generators, backtracking, ...
  - Compiler implementations
  - Web servers
Class exercises

E1. Describe the flow of control in the Standard ML expression

\[ f (2, g 3) \]

E2. (What does “flow of control mean”?)

E3. Describe the flow of control in

\[ f (g 2, h 3) \]

(Which “order of evaluation” did you choose?)
Continuations as the “rest of computation”

- A continuation is function value representing the rest of the computation,
  - with respect to a particular program,
  - with respect to a particular sub-expression,
  - with respect to a particular evaluation order.

(Think of a continuations as describing “what is going to happen to the value of this subexpression.”)
Example continuations

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Continuation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f (2, g 3) )</td>
<td>( fn \ v \Rightarrow f (2, v) )</td>
</tr>
<tr>
<td>( f (g 2, h 3) )</td>
<td>( fn \ v \Rightarrow f (v, h 3) )</td>
</tr>
<tr>
<td>( f (g 2, h 3) )</td>
<td>( fn \ v \Rightarrow v )</td>
</tr>
</tbody>
</table>

Notice that if \( P \) is the original program, \( E \) is the sub-expression, and \( K \) is the continuation, then \( K(E) \) gives the same result as \( P \).

(\( fn \ x \Rightarrow \ldots \) is the Standard ML notation for a function mapping \( x \) into \( \ldots \).)
Goal: Make the flow of control explicit and represent it using functions.

Purpose: When flow of control is represented as data, it can be manipulated. We will define some advanced control-flow operators.

Making continuations explicit:
- Each (non-trivial) expression takes its continuation as argument.
  ("Non-trivial" = may loop or may have side effects. Also called "serious.")
- Instead of returning a value, a serious expression passes this value to its continuation.
- The result of the entire program is passed to an (unknown) initial continuation.
CPS transformation by example

- Original program program:

  \[ f(2, g\ 3) \]

- CPS program:

  \[ fn\ k \Rightarrow\ g'\ 3\ (fn\ v \Rightarrow\ f'\ (2, v)\ k) \]

- Remember the rules:
  - Each serious expression takes its continuation as argument.
  - Each serious expression passes its result to its continuation.
  - The result of the entire program is passed to an (unknown) initial continuation.
Tips for the CPS transformation

- **Method**
  - Name all serious intermediate results. (Use let-expression if you like.)
    (Do not name expressions in tail position.)
  - Line up the serious expressions in the order they should be evaluated.
  - Use continuations to bind the intermediate results

- **Example expression:**

```
f(v1, v3)
```

```
let val v1 = g 1
    val v2 = g 2
    val v3 = g v2
in  f (v1, v3)
end
```
In the presence of this “special” continuation-passing style (CPS), we need a name for the “normal” style of programming. We call it *direct style* (DS).

We will mostly consider expressions, but CPS can also be defined for statements. (We can think of statements as special expressions that do not return a [meaningful] value.)

We need higher-order functions, so we implement our CPS programs in ML. Scheme would be an alternative. (And both Java and C# supports closures, but with a heavy notation.)
What is the result of CPS transforming
\[ f (g 1, h (2, 3)) \]

We have (at least) two options:

1. **Left-to-right:**
   \[
   \text{fn } k \Rightarrow \\
   g 1 (\text{fn } v1 \Rightarrow \\
   h (2, 3) (\text{fn } v2 \Rightarrow \\
   f (v1, v2) k))
   \]

2. **Right-to-left:**
   \[
   \text{fn } k \Rightarrow \\
   h (2, 3) (\text{fn } v2 \Rightarrow \\
   g 1 (\text{fn } v1 \Rightarrow \\
   f (v1, v2) k))
   \]

A CPS program must respect the evaluation order of the corresponding direct-style program.
A CPS program is *evaluation-order independent*: It gives the same result under left-to-right and under right-to-left.

(There are other variations of evaluation order that we do not consider here: call-by-value or call-by-need. They are also fixed by a CPS transformation.)
An example

- Direct style:
  ```plaintext
  fun double x = 2 * x
  fun twice (f, x) = f (f x)
  fun main y = twice (double, y)
  val result = main 5
  ```

- Continuation-passing style:
  ```plaintext
  fun double' x k = k (2 * x)
  fun twice' (f, x) k = f x (fn v => f v k)
  fun main' y k = twice' (double', y) k
  val result' = main' 5 (fn v => v)
  ```

- Notice that built-in (arithmetic) operators must be left in direct style.
E4. CPS transform these

```hs
fun add (x, y) = x + y
fun mul (x, y) = x * y
fun main y = add (mul (2, y), 3)
val result = main 7
```

E5. CPS transform the sum function

```hs
fun sum [] = 0
    | sum (x :: xs) = x + sum xs
```

(You may treat + as a direct-style primitive or use its CPS counterpart from the previous exercise.)

E6. CPS transform the map function (assuming the argument function f will be in CPS)

```hs
fun map f [] = []
    | map f (x :: xs) = f x :: map f xs
```
Practical CPS transformation

- We leave pattern matching and built-in operators in direct style.
- There is a choice between several ways of passing continuations to CPS transformed functions.
  
  \[
  g \ 2 \ (\text{fn} \ v \Rightarrow \ldots)
  
  g \ (2, \text{fn} \ v \Rightarrow \ldots)
  
  g \ (\text{fn} \ v \Rightarrow \ldots) \ 2
  
  g \ (\text{fn} \ v \Rightarrow \ldots, \ 2)
  \]

  For this course, the choice doesn’t matter as long as we are consistent. (We chose the first of the above, where the continuation is passed after the argument in a curried fashion.)
CPS transforming curried functions

Remember that f x y first applies f to x, and then applies the result (a function) to y. The expression is similar to

\[
\text{let } \text{val } g = f \ x \\
\text{in } g \ y \\
\text{end}
\]

and should therefore be CPS transformed to

\[
\text{fn } k \Rightarrow f \ x \ (\text{fn } g \Rightarrow g \ y \ k)
\]

but when we know that we’ll never partially apply f, we may instead CPS transform the expression to

\[
\text{fn } k \Rightarrow f \ x \ y \ k
\]
A left-to-right (call-by-value) CPS transformation

\[
\begin{align*}
[x_a]k &= kx \\
[e_1 e_2]k &= [e_1](fn \ v_1 => [e_2](fn \ v_2 => v_1 v_2 k)) \\
[fn \ x=e]k &= k(fn \ x => fn \ k' => [e]k') \\
[let \ x = e_1 \ in \ e_2]k &= [e_1](fn \ x => [e_2]k) \\
\end{align*}
\]
An interpreter is a means of expressing the meaning of one language (the interpreted language) in terms of a (possibly different) other language (the interpreting language).

We will present several interpreters showing how fundamental programming-language concepts can be expressed using continuations.
datatype exp = INT of int
    | ADD of exp * exp
    | SHOW of exp

fun eval (INT i) = i
| eval (ADD (e1, e2)) = eval e1 + eval e2
| eval (SHOW e) = 
    let val i = eval e
    in print (Int.toString i);
    i
end
Interpreter 1
(An evaluation-order independent interpreter)

datatype exp = INT of int
  | ADD of exp * exp
  | SHOW of exp

fun eval (INT i) k = k i
  | eval (ADD (e1, e2)) k
    = eval e2 (fn v2 => eval e1 (fn v1 => k (v1 + v2)))
  | eval (SHOW e) k
    = eval e (fn i =>
      ( print (Int.toString i);
        k i ))

fun run e = eval e (fn v => v)
E7. What is the difference between the following CPS version of

\[
\text{fun twice (f, x) = f (f x)}
\]

1.

\[
\text{fun twice (f, x) k = f x (fn v => f v k)}
\]

2. \[
\text{fun twice (f, x) k = f x (fn v => f v (fn v' => k v'))}
\]

E8. Related question: What is the difference between the following two (A-normal form) versions of \text{twice}:

\[
\begin{align*}
\text{fun twice' (f, x) = let val v1 = f x in f v1 end} \\
\text{fun twice'' (f, x) = let val v = f x val v' = f v in v' end}
\end{align*}
\]
Answer: The latter is not “tail-call optimized”: There is unnecessary binding of v’ requiring stack space.

The outermost call to f in twice is in “tail position”: It is the last thing the function does before it returns. This call does not need a stack frame: It can be optimized away.

Recursive function whose recursive calls are in tail position can be executing in constant space. (They are iterative.)

All of our interpreters will be tail-call optimized.
An interpreter almost in CPS, not capable of catching exceptions

An interpreter using two continuations, a *success* and a *failure* continuation:

At any point, control can flow to two different places: One for when evaluation completes normally and another for when evaluation terminates with an exception.
datatype exp = INT of int |
               ADD of exp * exp |
               FAIL

fun eval (INT i) k = k i |
                  eval (ADD (e1, e2)) k |
                        = eval e1 (fn v1 => eval e2 (fn v2 => k (v1 + v2))) |
                  eval (FAIL) k = 0

fun run e = eval e (fn v => v)
Interpreter 3
(Interpreter for exceptions, with catch)

datatype exp = INT of int
    | ADD of exp * exp
    | FAIL
    | TRY of exp * exp

fun eval e succ fail = ...

An expression receives two continuations:

- A success continuations, succ, invoked with a value when an expression terminates normally with that value
- A failure continuations, fail, invoked with no value (that is with argument ()) when an expression throws an exception (Notice that we interpret exceptions without using ML’s exceptions.)
The meaning of exceptions in Java

From the Java Language Specification, Third Edition:

14.20.1 Execution of try-catch

A try statement [...] is executed by first executing the try block. Then there is a choice:

- If execution of the try block completes normally, then no further action is taken and the try statement completes normally.
- If execution of the try block completes abruptly because of a throw of a value V, then [...]:
  - [...] the value V is assigned to the parameter of the selected catch clause, and the Block of that catch clause is executed. If that block completes normally, then the try statement completes normally; if that block completes abruptly for any reason, then the try statement completes abruptly for the same reason.
fun eval (INT i)    succ fail = succ i 
  | eval (ADD (e1, e2)) succ fail = 
  eval e1 (fn v1 =>
    eval e2 (fn v2 =>
      succ (v1 + v2))
    fail)
  fail
  | eval (FAIL)       succ fail = fail ()
  | eval (TRY (e1, e2)) succ fail = 
  eval e1 succ (fn () => eval e2 succ fail)

fun run e = eval e Int.toString (fn () => "failed!")
Local variables and global cells

We can add both local (immutable, ML-like) variables and global memory cells (and a combination of the two) to the interpreted language.

- Local variables requires an environment mapping variables to values.
  A local variables is introduced by a let expression, and is only visible in the body of this let expression. Variables cannot be assigned a new value.

- Global memory cells requires a store mapping cells to values.
  A (named) memory cell is introduced by an allocation, and is visible during the remaining execution. (Ah! That’s the continuation!) Memory cells can be assigned a new value.
Interpreter 4
(Interpreter with variables)

datatype exp = INT of int
  | ADD of exp * exp
  | VAR of string
  | LET of string * exp * exp

fun lookup env x = ...
fun extend env x v = ...

fun eval e env k = ...
fun eval (INT i) env k = k i
  | eval (VAR x) env k = k (lookup env x)
  | eval (ADD (e1, e2)) env k
      = eval e1 env (fn v1 =>
        eval e2 env (fn v2 =>
          k (v1 + v2)))
  | eval (LET (x, e1, e2)) env k =
      eval e1 env (fn v => eval e2 (extend env x v) k)

fun run e = eval e [] (fn v => v)

The environment passed to ADD(e1, e2) is passed unmodified to both e1 and e2. An expression does not change the environment.
datatype exp = INT of int  | ADD of exp * exp  
 | GET of string  | SET of string * exp  
 | SEQ of exp * exp

fun lookup sto x  = ...
fun update sto x v  = ...

fun eval e sto k = ... (* k receives new store *)
fun run e = eval e [] (fn (sto, v) => v)
fun eval (INT i) sto k = k (sto, i)
| eval (GET x) sto k = k (sto, lookup sto x)
| eval (ADD (e1, e2)) sto k
  = eval e1 sto (fn (sto, v1) =>
    eval e2 sto (fn (sto, v2) =>
      k (sto, v1 + v2)))
| eval (SET (x, e1)) sto k =
  eval e1 sto (fn (sto, v) => k (update sto x v, v))
| eval (SEQ (e1, e2)) sto k =
  eval e1 sto (fn (sto, v) => eval e2 sto k)

The store passed to ADD(e1, e2) is passed to e1. e1 produces a
new store which is passed to e2. The store produced by e2 is the
store produced by ADD(e1, e2).
CPS interacts with other language features in the interpreter.

Some questions that we must address:

■ Which syntactic category may raise exceptions?
■ Which syntactic category may side effect the store?
■ Which syntactic category have access to the store?
■ What is the scope of local definitions?
■ How does exception (and other control operators) interact with side effects on the store?
Defining CPS interpreters systematically

- Define syntax inductively
- Define evaluation functions inductively
  - Pass an environment to access local variables.
  - Pass a store to access the store.
  - Pass a (success) continuation to model “normal completion.”
    - Pass a result to this continuation when values are produced upon normal completion.
    - Pass a store to this continuation when the store may be modified during normal completion.
  - Pass a failure continuation to model “exceptional (or abrupt) completion”
    - Pass a result to this continuation when values are produced upon exceptional completion.
    - Pass a store to this continuation when the store may be modified during exceptional completion.
Define abstract syntax trees inductively:

- Each syntactic category $S$ is defined by a separate datatype:
  
  ```plaintext
datatype S = ...
  ```

- If $S$ may contain subelements of another syntactic category $T$, then $S$ uses the datatype of that syntactic category
  
  ```plaintext
datatype T = ...
datatype S = ... of T | ...
  ```

- Example
  
  ```plaintext
datatype exp = INT of int | ADD of exp * exp
datatype stm = WHILE of exp * stm | ...
  ```
Define evaluation functions inductively:

- The evaluation (or execution) of each syntactic category $S$ is defined by separate function:

  ```
  fun eval_S s ... = ...
  ```

  This function is defined inductively by pattern matching over the datatype $S$.

- If $S$ may contain subelements of another syntactic category $T$, then $\text{eval}_S$ uses the evaluation function of that syntactic category

  ```
  fun eval e ... = ...
  
  fun exec (WHILE (e, s)) ... = ... eval_exp e
  ```
Defining CPS interpreters systematically

(Local variables)

An environment maps (names of) local variables to (denotable) values:

```
val empty = []
fun extend env x v = (x, v) :: env
fun lookup [] x
    = raise Fail "Unbound variable"
| lookup ((y, v) :: env) x
    = if x=y then v else lookup env x
```
Defining CPS interpreters systematically

(Local variables, continued)

Environments are

- accessible to syntactic categories containing local variables;
- augmented temporarily by constructs that define new local variables;
- not updated.

```ml
fun eval (VAR x) env k
    = k (lookup (env, x))

| eval (LET (x, e1, e2)) env k
    = eval e1 env (fn v1 =>
        eval e2 (extend (env, x, v1)) k)

| eval (ADD (e1, e2)) env k
    = eval e1 env (fn v1 =>
        eval e2 env (fn v2 => k (v1 + v2)))
```
A store maps (names of) memory cells to (storable) values:

```ml
val init = []
fun lookup [] x = raise Fail "Unknown cell"
  | lookup ((y, v) :: sto) x
      = if x = y then v else lookup sto x
fun update [] x v = [(x, v)]
  | update ((y, w) :: sto) x v =
      if x = y then (x, v) :: sto
      else (y, w) :: update sto x v
```
The store is

- accessible to syntactic categories referencing memory cells;
- produced by syntactic categories that may update memory cells.

```
fun exec (ASSIGN (x, e)) sto k
    = eval e sto (fn v => k (update sto x v))
  | exec (SEQ (s1, s2)) sto k
    = exec s1 sto (fn sto =>
        exec s2 sto k)
```