Case Study: Continuation Semantics
Fake imperative programming using CPS, dictionaries, and datatypes
In this lecture, I use a lot of code and ideas developed by others.

- Red/Black Trees for dictionaries: code by Mike Erdmann and Frank Pfenning
  - We’ll discuss the details of this next week!
- Monadic Parser Combinators: core parser code by Matthew McQuaid, for the course 98-317 (spring 2020)
- Continuation Semantics for while programs: I based my code off of notes and lectures by Steve Brookes for the course 15-314/812 (spring 2020)
(code demo)
It takes a couple steps to do this.

1. Represent the FC code in a syntax SML can understand
2. Design a mechanism for how to mimic mutable state in SML
3. Write (CPS!) functions which “run” the SML representation of the FC code

The first step is more involved (and sophisticated) than we can get into here, so we’ll mainly focus on the latter two steps.
The files in red are library code, which you don’t need to worry about.

- cExp.sml – the SML syntax of FC
- FC.sml – the core logic
- *.fc – example files (written in FC)
- Makefile – allows you to run `make repl` to start an smlnj repl with everything needed to run .fc files
- lib
  - parse.sml – code for parsing FC to its SML representation
  - Dictionary.sml – code for dictionaries
  - sources.cm – info for SMLNJ to let it know what files to load
Syntax for running code

In your terminal shell:

```bash
make repl
Standard ML of New Jersey v110...
...
[New bindings added.]
- FC.Runfile "filename.fc";
```
Section 1

Representing FC programs in SML
The three expression types

We represent FC programs using three datatypes:

- **cExp**: represents commands. The program as a whole is represented by a value of type `cExp`. These are built up from some basic commands via various operations.
- **iExp**: represents integer expressions, which could be a variable name, an integer constant, or various arithmetic combinations of other integer expressions.
- **bExp**: represents boolean expressions, which could be a variable name, a boolean constant, boolean operations on other boolean expressions, or comparisons between integer expressions.
iExp = iVAR of string
|  | CONST of int
|  | PLUS of iExp * iExp
|  | TIMES of iExp * iExp
|  | NEG of iExp
|  | DIV of iExp * iExp
datatype bExp = bVAR of string
    | TRUE
    | FALSE
    | EQ of iExp * iExp
    | LT of iExp * iExp
    | GT of iExp * iExp
    | AND of bExp * bExp
    | NOT of bExp
    | OR of bExp * bExp
13.2

datatype cExp = SKIP
| ASSIGNB of string * bExp
| ASSIGNI of string * iExp
| THEN of cExp * cExp
| IFTHENELSE of bExp* cExp *cExp
| WHILE of bExp * cExp
| RETURN of iExp
We’ve written some code which

1. Reads the .fc file
2. Builds a single value of type cExp representing the code

```ocaml
(* Accepts a string of FC code and parses it *)
val fcParser.parse : string -> (cExp -> 'a) -> (unit -> 'a) -> 'a

(* Accepts a filename and reads FC code in it *)
val fcParser.fileParse : string -> (cExp -> 'a) -> (unit -> 'a) -> 'a
val fcParser.showParse : string -> cExp
```

**Note:** The parser is currently somewhat buggy. I’m working on improving it.

**Note:** Parsing is a really interesting topic. Learn more about it if you get the chance!
Steps

✓ Represent the FC code in a syntax SML can understand
2 Design a mechanism for how to mimic mutable state in SML
3 Write (CPS!) functions which “run” the SML representation of the FC code
Section 2

Dictionaries
A dictionary is a data structure which stores key-value pairs \((k, v)\), which can be looked up (i.e. you supply a string \(k\), and the dictionary tells you the corresponding value \(v\), if there is one). We implement dictionaries to have the following methods.

\[
\begin{align*}
\text{Dict}.\text{empty} & : \text{ 'a Dict}.\text{dict} \\
\text{Dict}.\text{lookup} & : \text{ 'a Dict}.\text{dict} \to \text{ string } \to \text{ 'a option} \\
\text{Dict}.\text{insert} & : \text{ 'a Dict}.\text{dict} \to (\text{ string } \times \text{ 'a}) \\
& \quad \to \text{ 'a Dict}.\text{dict}
\end{align*}
\]

We won’t concern ourselves with the implementation details today – we leave that for another time! Just assume these dictionaries work as intended.
Using dictionaries to mimic mutable state

We want to simulate a “mutable state”, where variables are set to certain values and can be modified later. We also want to be able to allocate arbitrarily-named variables to have either boolean or integer values. We can do this by passing a dictionary $D : t \ 	ext{Dict.dict}$ around:

- All our functions will take in a dictionary as an argument, representing the “current state”
- Set a variable $x$ to $v : t$ by putting
  
  ```haskell
  val D' = Dict.insert D ("x",v)
  ```

  and then using $D'$ as the state from then on (e.g. passing to other functions)
- Query the current value of $x$ by putting
  
  ```haskell
  val xVal = Dict.lookup D "x"
  ```

  If $xVal$ is $\text{SOME} \ v$ then $x$ is currently set to $v$. If $xVal$ is $\text{NONE}$, then $x$ is currently unbound.
How we’ll keep track of variables

13.3

```ocaml
datatype entry = BOOL of bool | INT of int
```

So an `entry Dict.dict` stores booleans and integers, tagged with their types.

- If `Dict.lookup D "x"` is `SOME (BOOL b)`, then `x` is set a boolean-valued variable, whose value is currently `b`.
- If `Dict.lookup D "x"` is `SOME (INT n)`, then `x` is an integer-valued variable whose current value is `n`. 
Steps

✓ Represent the FC code in a syntax SML can understand
✓ Design a mechanism for how to mimic mutable state in SML
3 Write (CPS!) functions which “run” the SML representation of the FC code
Section 3

Execution
A system of errors

13.4

```haskell
datatype Type = Bool | Int

datatype error =
    TypeError of string * Type * Type
    | UnboundVar of string
    | DivZero
    | NoReturn
```
interpret : cExp -> (error -> 'a) -> (int -> 'a) -> 'a

REQUIRES: true

ENSURES: interpret input panic success evaluates to success (n) if executing the command input returns n. If executing input encounters an error e, then interpret input panic success evaluates to panic e.
fun interpret input panic success =
  let
    fun evalB (D:entry Dict.dict) (b:bExp) (k:bool -> 'a) : 'a = ...
    fun evalI (D:entry Dict.dict) (e:iExp) (k:int -> 'a) : 'a = ...
    fun exec (D:entry Dict.dict) (c:cExp) (k:entry Dict.dict -> 'a):'a = ...
evalI is CPS to the core!

fun evalI (D: entry Dict.dict) (e: iExp) (k:int -> 'a) =

  case e of
    (CONST n) => k n
    | (PLUS(e1,e2)) =>
      evalI D e1 (fn v1 =>
        evalI D e2 (fn v2 =>
          k(v1+v2)))
    | (TIMES(e1,e2)) =>
      evalI D e1 (fn v1 =>
        evalI D e2 (fn v2 =>
          k(v1*v2)))
evalI is CPS to the core!

### 13.6

```
| (NEG(e')) => evalI D e' (fn v => k(¬v))
| (DIV(e1,e2)) => evalI D e2 (fn 0 => panic DivZero
| v2 => evalI D e1 (fn v1 => k(v1 div v2)))
| (iVAR i) =>
  (case (Dict.lookup D i) of
    (SOME(INT v)) => k v
    | (SOME _) => panic (TypeError (i,Int,Bool))
    | NONE => panic (UnboundVar i))
```
and so is evalB!

```haskell
fun evalB (D:entry Dict.dict) (b:bExp) (k:bool->'a) =
  case b of
    TRUE => k true
  | FALSE => k false
  | (EQ(e1,e2)) =>
      evalI D e1 (fn v1 =>
        evalI D e2 (fn v2 =>
          k(v1=v2)))
  | (LT(e1,e2)) =>
      evalI D e1 (fn v1 =>
        evalI D e2 (fn v2 =>
          k(v1<v2)))
  | (GT(e1,e2)) =>
      evalI D e1 (fn v1 =>
        evalI D e2 (fn v2 =>
          k(v1>v2)))
```

Jacob Neumann
Case Study: Continuation Semantics
05 June 2020 25 / 30
and so is evalB!

```
| (OR(b1, b2)) =>
  evalB D b1 (fn v1 =>
    evalB D b2 (fn v2 =>
      k(v1 orelse v2)))
| (NOT(b')) =>
  evalB D b' (fn v => k (not v))
| (bVAR(i)) =>
  (case (Dict.lookup D i) of
    (SOME(BOOL v)) => k v
    | (SOME(INT _)) =>
      panic (TypeError(i, Bool, Int))
    | NONE =>
      panic (UnboundVar i))
```
fun exec (D : entry Dict.dict) (c : cExp) (k : entry Dict.dict -> 'a) : 'a =
    case c of
      SKIP => k D
    | (ASSIGNB (s, b)) =>
        evalB D b (fn vb =>
            k (Dict.insert (D, (s, BOOL vb))))
    | (ASSIGNI (i, e)) =>
        evalI D e (fn v =>
            k (Dict.insert (D, (i, INT v))))
and finally exec

| (THEN(c1,c2)) => exec D c1 (fn D' => exec D' c2 k) |
| (IFTHENELSE(b,c1,c2)) => evalB D b (fn true => exec D c1 k | false => exec D c2 k) |
| (WHILE(b,c')) => evalB D b (fn true => exec D c' (fn D' => exec D' c k | false => k D)) |
| (RETURN e) => evalI D e success |
fun interpret input panic success =
  let
    fun evalB (D:entry Dict.dict) (b:bExp) (k:boolean -> 'a) : 'a = ...
    fun evalI (D:entry Dict.dict) (e:iExp) (k:int -> 'a) : 'a = ...
    fun exec (D:entry Dict.dict) (c:cExp) (k:entry Dict.dict -> 'a) : 'a = ...
      (* calls success to return*)
  in
    exec (Dict.empty) input (fn _ => panic NoReturn)
  end
Thank you!