Functional languages
Arithmetic expressions and functions

\[32 \times (51 + 1)\]

\[\text{let } f x = (3 \times x)\]
\[\text{let } f = \text{fun } x \rightarrow (3 \times x)\]
\[f(12) + 2\]

\[\text{let } g x y = 3 \times x + y\]
\[\text{let } g = \text{fun } x \rightarrow (\text{fun } y \rightarrow 3 \times x + y)\]
\[8 + (g 33 5)\]
Functions on the right (functions as arguments)

let g = fun f -> f 3
let h = fun x -> x+5

\[
g \circ h
\]

let p = fun x y -> x+y
let q = fun z -> z+2

\[
g \circ p \ 8
\]

\[
p \ 7 \ (q \ 3)
\]
Typical programs we want to execute, and how we write them

- notation for applications: \( g \ 3 \)
  - in maths: \( g(3) \)
  - sometimes \( g@3 \) to stress that application is a binary operator

- using the \texttt{let} construct

  - a program is a sequence of \texttt{lets},
    possibly followed by an expression (the “main”)

- \texttt{let x = 3 in let y = 4 in let z = 5 in (x+y*z)}

  will also be written

  
  \begin{align*}
  \begin{align}
  \text{let } & x = 3 \\
  \text{let } & y = 4 \\
  \text{let } & z = 5 \\
  (x+y*z) \\
  \end{align}
  \end{align*}

- a \texttt{nested} \quad \texttt{let..in}

  \begin{align*}
  \begin{align}
  \text{let } & f = \text{fun } x \rightarrow \\
  \text{let } & y = g (x*x) \text{ in} \\
  \text{if } & y>0 \text{ then } y \text{ else } x \\
  \end{align}
  \end{align*}

  \begin{itemize}
  \item here is \( f \)'s return (\( y \) or \( x \)) 
  \end{itemize}
**Fun**, a small functional programming language

- **syntax**

  ```
  e ::= fun x → e | e₁ e₂ | x ➤ core functional language
       | let x = e₁ in e₂ ➤ if you insist
       | e₁ + e₂ | 1, 2, 3, ...
  ```

  \( x, y, z, \ldots \in \mathcal{V} \)  ➤ variable identifiers

- **two versions of the operational semantics**

  - **first version**: \( e \Downarrow \nu \) ➤ *no environment*
  - **second version**: \( \sigma, e \Downarrow \nu \)

  **Demo** ➤ see also the (flawed) implementation
Compiling to an Abstract Machine

on the board
Program transformations in Fun
The reason for closures

- recall the example that motivated the introduction of closures

```plaintext
let h = fun t -> t+t
let g = fun y -> 30 + (h y)
let h = 12
  g 5

- hence the Abstract Machine transition

\[ \text{Closure}(x,c'); c \mid \sigma \mid s \parallel c \mid \sigma \mid (x,c')[\sigma].s \]

notice the duplication of the environment
Free and bound variables in programs

- a Fun program

```
let t = 3
let u = fun x -> x*2
let v = fun z -> z + u (2*z)
v t + u 12
```

```
let g = fun x y ->
  let z = x+2*t in
  z + (f y)
```

A definition (like the one for \(g\) above) makes sense provided the variables it uses make sense in the environment where the definition occurs.

- in `let x = e_1 in e_2`, \(x\) is bound in \(e_2\)
- in `fun x -> e`, \(x\) is bound in \(e\)

A variable is free if it is not bound. 

- nota: `let x = e_1 in e_2` behaves like `(fun x -> e_2) e_1`
- `let x = e_1 in e_2` and `fun x -> e` are binders, the scope of \(x\) is \(e_2\) (resp. \(e\))

**scope is dope / static scope is extatic dope**
Handling closures

- back to the example

```ml
let h = fun t -> t+t
let g = fun y -> 30 + (h y)  g = (fun y -> 30 + (h y))[(h, fun t -> t)]
let h = 12
  g 5
```

- in compilers for functional languages, closures are typically represented by a pair
  1. pointer to the code for the body
  2. pointer to the environment

  - has to be allocated in the heap
  - not the whole environment
  - may contain, recursively, other pointers to environments
Representing closures: closure conversion

- represent *explicitly* closures in the language
  
  Fun extended with tuples/records/structs

- modify functions:
  - when they are defined

\[
\begin{array}{l}
\text{let code = fun } (c,x) \rightarrow \\
\quad \text{let } (_,x_1,...,x_n) = c \text{ in } [e] \\
\quad \text{in (code, } x_1,...,x_n)
\end{array}
\]

where \(x_1,...,x_n\) are the free variables of \(\text{fun } x \rightarrow e\)

- ”let \((_,x_1,...,x_n) = c \text{ in } [e]”": the function reconstructs the environment before executing \([e]\)

- and when they are called

\[
\begin{array}{l}
\text{let } c = [e1] \text{ in } \\
\quad \text{let code = proj}_0(c) \text{ in } \\
\quad \text{code (c, [e2])}
\end{array}
\]
Lambda lifting: a transformation from Fun to Fun

- transforming the program in order to obtain a flat structure for functions
- pulling out functions defined within other functions

(example:)

```ocaml
define f x =
define g y = x+y in
g 5*x + g 3*x
```

- modifying the definition and the calls to these functions

(modifying g above)

- we obtain a set of recursive definitions of functions,
  - with no free variable
  - all at the same level
Continuations
Introducing continuations

- replace “returns” with function calls
  from  `let f x y = x+2*y`
  to  `let f x y k = k (x+2*y)`
  `k` is the “future” of the computation

- calling a function  `let f x y = y + (g (2*x))`
  - first compute `g (2*x)`
  - then (*return inside f and*) add `y`

  `let f x y k =`
  `let k’ = fun u -> k (y + u) in`
  `g (2*x) k’`  (*`k’ is the future of the computation of g (2*x)*)

- recursive functions  `let rec fact n = if n<2`
  `then 1 else n*(fact (n-1))`

  `let rec fact n k = if n<2 then k 1`
  `else let k’ = fun u -> k (n*u) in fact (n-1) k’`
Making it systematic: the CPS translation

CPS: Continuation Passing Style

► every value is translated into a program that waits for a receiver for this value

\[ 12 \] = \text{fun } k \rightarrow k \ 12 \quad k: \text{the receiver} 

NB: using “k” for continuations is rather standard, let’s forget about using \( k \) for integers (\( \in \mathbb{Z} \))

► accordingly, define a translation from Fun to Fun,

► written \([e]\)

► obeying the CPS convention: \([e] = \text{fun } k \rightarrow \ldots\)
Continuations and control

CPS yields a style in which function calls express all forms of control-flow

▶ the flow is **explicit**
  ▶ “fun v -> ..” insures sequentialisation
  ▶ for instance, you know which summand you evaluate first

▶ while loops **Demo** do_while_cont.ml
  but also: return, break, continue, for

▶ exceptions
  ▶ try with / raise, try catch / throw
  ▶ it is possible to translate Fun+exceptions into Fun
Properties of the CPS translation on the board
Tail calls

- consider \( \text{let } f \ x = g(h(x)) \)
  - first call \( h \), then return in \( f \)
  - then call \( g \) ← tail call
  - then return in \( f \), and exit from \( f \)

- tail calls can be compiled in a specialised way, so that we exit from \( f \) when calling \( g \)
  - no push on the stack

- tail recursive functions: recursive calls are tail calls
  - the stack does not grow along recursion

DEMO append.ml, term.ml
CPS as an intermediate representation

the target language of the transformation is almost an intermediate language

- every call is terminal
  in principle, no need for a stack  
  
  \( \text{always one function alive} \)

- refining the CPS transform to yield simpler programs
  - “administrative” reductions
  - treatment of arithmetical expressions
    when there are no function calls
  - treating n-ary functions as such
    do not translate \( \text{fun } x_1 \rightarrow \text{fun } x_2 \rightarrow \text{fun } x_3 \rightarrow e \)
    to \( \text{fun } x_1 \rightarrow \text{fun } k_1 \rightarrow \text{fun } x_2 \rightarrow \text{fun } k_2 \)
    \( \rightarrow \text{fun } x_3 \rightarrow \text{fun } k_3 \rightarrow [e] \)
    but to \( \text{fun } x_1 \ x_2 \ x_3 \ k \rightarrow [e] \)
    or maybe to \( \text{fun } (x_1, x_2, x_3, k) \rightarrow [e] \)
  - distinguishing “true functions” from continuations (jumps)

- backwards transformation (out of CPS)
  - in order to compile using a stack
  - CPS form used for optimisation purposes
CPS vs SSA

- the CPS transform yields programs which
  - are rather difficult to read
  - involve elementary operations
    - arithmetic operations and function calls only on atoms (variables, constants)
    - function calls are terminal

- CPS form (and its refinements/variations/improvements) is used as an intermediate representation for functional compilers

- CPS: the functional counterpart of SSA
  - unique assignment to variables
  - dominators $\leftrightarrow$ scope
  - $\varphi$ nodes correspond to (some) continuations
    - join point in the CFG $\leftrightarrow$ continuation
    - transfer of control and expressing $\varphi$ $\leftrightarrow$ calling a continuation