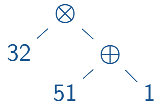
Functional languages

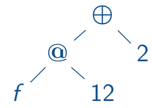
Arithmetic expressions and functions

32*(51+1)



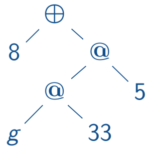

```
let f x = (3*x)
let f = fun x -> (3*x)
```

f(12)+2




```
let g x y = 3*x+y
let g = fun x -> (fun y -> 3*x+y)
```

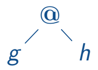
8 + (g 33 5)

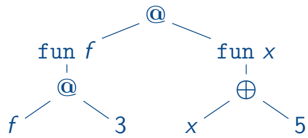


Functions on the right (functions as arguments)

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

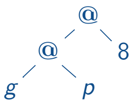
g h



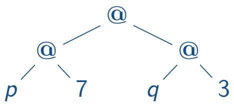



```
let p = fun x y -> x+y
let q = fun z -> z+2
```

g 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 `let` construct
  - a program is a sequence of `let`s, possibly followed by an expression (the "main")
  - `let x = 3 in let y = 4 in let z = 5 in (x+y*z)`

    will also be written
    ```
    let x = 3
    let y = 4
    let z = 5
    (x+y*z)
    ```
  - a **nested** `let..in`
    ```
    let f = fun x →
      let y = g (x*x) in
    if y>0 then y else x
    ```
    ⟵ here is f's `return` (y or x)

Fun, a small functional programming language

- syntax

$$e \;::=\; \texttt{fun}\ x \to e \mid e_1\ e_2 \mid x \quad \text{core functional language}$$
$$\mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$$
$$\mid e_1 + e_2 \mid 1, 2, 3, \ldots \quad \text{if you insist}$$

$x, y, z, \ldots \in \mathcal{V}$ variable identifiers

- two versions of the operational semantics

on the board

- first version: $e \Downarrow v$ *no environment*
- second version: $\sigma, e \Downarrow v$

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

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

- hence the Abstract Machine transition

Closure(x,c'); c | σ | s ‖ c | σ | (x,c')[σ].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 x₁ = e₁ in
let x₂ = e₂ in
...
let x_k = e_k in
e
```

```
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 = e1 in e2`, `x` is **bound** in `e2`
  in `fun x -> e`, `x` is **bound** in `e`
  a variable is **free** if it is not bound *"free", or "non local"*
  - nota: `let x = e1 in e2` behaves like `(fun x -> e2) e1`
  - `let x = e1 in e2` and `fun x -> e` are *binders*, the *scope* of x is `e2` (resp. `e`)

scope is dope / static scope is extatic dope

Handling closures

- back to the example

```
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 DEMO `clos.ml`
     - 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

```
[fun x -> e] =  let code = fun (c,x) ->
                  let (_,x1,...,xn) = c in [e]
                in (code, x1,...,xn)
```

where `x1,...,xn` are the free variables of `fun x -> e`

  - "`let (_,x1,...,xn) = c in [e]`": the function reconstructs the environment before executing `[e]`

  - and when they are called

```
[e1 e2] =  let c = [e1] in
           let code = proj₀(c) in
           code (c,[e2])
```

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
  (in the "e" of a `fun x -> e`))

example:

```
let f x =
   let g y = x+y in
   g 5*x + g 3*x

let g' y x = x+y
let f' x = g' x 5*x + g' x 3*x
```

- modifying the definition and the calls to these functions
  (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] = fun k -> k 12`   k: 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] = fun k -> ..`

on the board

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 `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 *(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 `fun x1 -> fun x2 -> fun x3 -> e`
    to `fun x1 -> fun k1 -> fun x2 -> fun k2 -> fun x3 -> fun k3 -> [e]`
    but to `fun x1 x2 x3 k -> [e]`
    or maybe to `fun (x1, x2, x3, k) -> [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 ↔ scope
  - φ nodes correspond to (some) continuations
    - join point in the CFG ↔ continuation
    - transfer of control and expressing φ ↔ calling a continuation