Functional Languages

Overview

- Lambda calculus
- Elements of functional programming
  - Expressions
  - Control flow
  - Scoping
- Dialects of functional programming languages
- Lasting contribution of functional languages

Historical Background

- Head of “Turing’s model of computation” and/or ”Turing machine”?
- If you have taken theory of computation, you’ve heard about “tapes” and moving “left/right”
- You probably also heard/learned about “Church’s thesis” which is roughly that models of computation are equally powerful
- Turing machines are an (imperative) abstract model of computation that (vastly) simplifies computers:
  - Left/right relates to addresses and data-movement
  - Writing/erasing the tape is a simplification of updating memory: load/store with in-between computations
  - Unbounded tapes basically mean that we don’t worry about memory, i.e. that our program will fit in memory no matter what (in practice we “just make them fit” or limit our programs to run on some hardware somehow)
- Church’s thesis uses another model of computation: *Lambda Calculus*

Lambda Calculus

- Abstract model of computation
- Informally, it defines functions, free variables and bound variables
- Examples of lambda functions:
  - λ x . x == x → x (identity)
  - λ x . y == x → y (constant)
  - λ x . x + y == x → x + y (add input x to a variable y not yet defined)

Lambda Calculus

- Produce lambda expressions via lambda terms (λ-term), which are created with the following inductive rules:
  - a) Variables
  - b) An abstraction: If *t* is a λ-term, and *x* a variable, then (λ*x*.*t*) is a lambda term
  - c) An application: If *t* and *s* are λ-terms, then (*ts*) is also a lambda term
- In imperative terms, (a) above defines a variable, (b) defines how functions are created (given argument x, return expression t), (c) defines composition of functions:

  λ x . x + y == x → x + y (add input x to a variable y not yet defined)

Functional Languages

- Lack state
- Based on functions and expressions:
  - no explicit assignment
  - no functions with side-effects
  - no updating memory
- All the above translates to: no dependences and a lot of freedom to reorder program operations
- It's a more "mathy" world:
  - iterative constructs → recursion
  - Multi-dimensional arrays → nested lists

- Considers functions as First Class Citizen
- An object/construct is a First Class Citizen in a Programming Language if it can be:
  - Returned as a result of subroutine (even as a parameter)
  - Passed as a parameter to a subroutine
- So Functional languages consider functions as first class citizens
- Many languages provide lambda support (e.g. C++ since version 2011, Java 8)

Examples of Functional Languages

- Lisp: 1958, second oldest programming language, only Fortran is older, stands for ”LISP Processor”, has *MANY* descendants
- Common Lisp: early 80s, language specification for Lisp, general purpose, multi-paradigm, procedural, functional and object-oriented language
- Scheme: created in the 70s, used lexical scope, used tail-call optimization, first-class continuations (data structures representing subroutine execution state and accessible by the programmer)
- ML (Meta Language): call-by-value evaluation, static typing, type inference, static scoping, currying
- OCaml (Objective Caml): Caml being a dialect of ML, offers object oriented features, byte code compiler, native code optimizer, reversible debugger
- Haskell: early 90s, purely functional, statically typed, with type inference and lazy evaluation
- Scala: mid 2000s, strong type system, object oriented, meant to be compiled with Java
- Rust: 2010, concurrent and safe programming language, automatic memory management, multi-paradigm, C-inspired syntax
- Erlang: introduced in mid/late 80s, concurrent functional language, distributed and fault-tolerant run-time system, hot-swapping (code update without powering down system)

Scheme Arithmetic

- `7 ➔ constant 7`
- `(+ 3 4) ➔ adds 3 and 4`
- `(+ x y) ➔ adds x and y`
- `(+ (* x y) z) == (x * y) + z`
- `(define x 3) ➔ will define x=3, type x next`
- `x ; will print 3`

In lists, the first element is always expected to be the name of an operator/function

Note that x and y are not defined (yet)

These define x and y, now we can try summing them again

Evaluation

- Scheme/Lisp functional languages evaluate expressions
- Languages provide mechanism to force and to prevent evaluation
- Evaluation is the core of a functional language, see eval
- Normally, an expression in parenthesis will be automatically evaluated:

  `(+ 3 4)`

- The language will assume that the first element in the list is the operation name, so this is an error:

  `((7))`

- Quote operation prevents evaluation:

  `(quote (+ 3 4))` ➔ `(+ 3 4)`

  `(‘ (+ 3 4))` ➔ `(+ 3 4)`

Types and Inspection

- Usually type-check deferred to runtime
- Every expression has a type (as with imperative languages)
- Languages provide mechanisms to inspect type:

```
(boolean? x)    ;  is x a Boolean
(char? x)     ; is x a character
(string? x)
(symbol? x)  ; akin to an identifier, but not exactly, wider interpretation
(number? x)
(pair? x)
(list? x)
```

- Return (anonymous) functions:

```
(lambda (x) (* x x))    ; this is a function
```

Scheme Lists

A lot of list manipulation

- Car (head of list):

```
(car '(2 3 4)) == 2
```

- Cdr (input list minus head):

```
(cdr '(2 3 4)) == (3 4)
```

- Cons (place element/item as front element to list):

```
(cons 2 '(3 4)) == (2 3 4)
```

- Cadr (car(cdr)):

```
(cadr '(1 2 3 4 5)) == 2
```

- Caddr (car (cdr (cdr))):

```
(caddr '(1 2 3 4 5)) == 3
```

- Caadr doesn't exist

Scheme Lists

- Empty list:

```
()              ;    (nil)
```

- Inserting 5 in the front of the empty list:

```
(cons 5 ())   ;    (5, cdr) → (nil)
```

- Inserting 2 in the front of the list containing only 2:

```
(cons 2 (cons 5 ())    ;  (2,cdr) → (5,cdr) → (nil)
```

- Which is not the same as:

```
(cons 2 5)  ; this is a pair, not a list, equiv to   (2,5) and not (2,cdr)→(nil)
```

- A shorthand for lists:

```
(list 5 2 3)
```

Scheme Lists

- Concatenating two lists:

```
(append ‘(1 2) ‘(3 4));   (1 2 3 4)
(append ‘1 ’(2 3)); ERROR
(append 1 ’(2 3)); ERROR
```

- Replace head of list:

```
(cons 5 (cdr ‘(1 2 3 4 5))); (5 2 3 4 5)
```

- Reversing lists:

```
(reverse ‘(1 2 3)); (3 2 1)
```

Scheme Predicates

- Checking whether a list has no elements:

```
(null? ()); #t
(null? '(1 2 3)); #f
```

- Comparing if two lists are "equal":

```
(equal? (list 2 3 4) (cons 2 (cons 3 (cons 4 ()))));  #t
```

- Comparing numbers:

```
(eq 3 3.0);  #t
(eq 3 3.01); #f
(eq 3 3.000000000000000000001); #t
```

- Type predicates:

```
(number? 3); #t
(number? 3.0); #t
(number? "three"); #f
```

Quick Scheme Arithmetic

- Usual operators:

```
(+ 3 4); 7
(* 2 5); 10
(- 2 4); -2
(/ 5 2);  5/2
(/ 5.0 2); 2.5
(/ 5 2.0); 2.5
```

- Remainder/modulo:

```
(remainder 5 2); 1
(remainder 10 4); 2
(remainder 10.2 4);  ERROR
```

A bit more of Lambda

- Lambda creates a function, for example

```
(lambda (x) (* x x))  ; function
```

- The above:
  - Creates an anonymous function, with a function body (* x x)
  - The operation is “lambda” and has two (list) arguments: (x) and (* x x)
  - First list argument to lambda is the list of arguments to be passed to the anonymous function when it is called
  - The second list argument to lambda is the body of the anonymous function

A bit more of Lambda

- Lambda creates a function, which can then ***be applied*** for example

```
((lambda (x) (* x x) 3)  ; function
```

- The above creates an anonymous function and then applies it to 3
- We don’t say ”pass 3 to the anonymous function” we say “we create the anonymous function XYZ and apply it to 3”
- Internally:
  - the ***eval*** function creates a new activation record, adds space for all its inputs and outputs
  - bindings of formal parameters to referencing environment (all the possible active subroutines)
  - Expression arguments of the body evaluated in order
  - By convention, last evaluated expression is the result of the function

(Nested) Scopes: `let`

- Allows to ***bind*** names to functions
- Try these:

```
(let (x 3) x)                    ; evaluates to 3
(let (x 3) (+ x 0)               ; evaluates to 3
(let ((x 3) (y 4)) (+ x y))      ; evaluates to 7
```

- Informally, the above creates a name ”x”, bounds it to 3 (sort of initializes) and then uses it in some function with some value
- `let` also creates a ***scope*** for variables
- In imperative functional languages, we just define functions as operations, and assume arguments come with some value
- In functional languages:
  - we have to say “you have to use this value in this function”
  - Values can come from further, outward-nested “let” scopes

(Nested) Scopes: `let`

- **`let`** takes two or more arguments:
  - Argument 1 is a list of pairs; each pair is an identifier and an expression to bind it to
  - Each argument K (K >= 2) is evaluated with the values of the first list

```
(let ((a 3))          ;; a list with a single pair
   (let
     ((a 4) (b a))
       (+ a b)
   )
)
```

- Binds a new ‘a’ to 4; Binds ‘b’ to outer ‘a’; computes ‘outer a’ plus ‘b’
- The above happens after binding an ‘a’ to 3
- Final result?

(Nested) Scopes: **`let`**

- Another example

```
(let ((a 3)
      (b 4)
      (square (lambda (x) (* x x)))
      (plus +))
   (sqrt (plus (square a) (square b))))
     ➔ 5.0
```

Control Flow and Conditionals

```
(if (< 2 3) 10 20); prints/returns 10
(if (< 4 3) 10 20); prints/retruns 20
(if (< 2 3) (+ 2 3) (* 2 3)); prints 5
(if (> 2 0) (+ 2 3) (+ 2 “foo”)); ERROR

(cond
  ((< 3 2) 1)
  ((< 4 3) 2)
  (else 3))  ; 3
```

Recursive Functions

- **`let`** has issues with recursive functions because names are binded only for nested scope (or nested lets)
- Functional Languages (e.g. Scheme) provide a special **`let`** with nested semantics that allow to use binded names “at the same level of nesting”:

```
(letrec ((fact
  (lambda (n)
     (if (= n 1) 1
         (* n (fact (- n 1)))))))
  (fact 5))
```

➔ 120

Define: A "Global" **`let`**

- Scheme, Lisp and Common Lisp use static scopes
- Most other Lisp dialects use dynamic scopes
- Both **`let`** and **`letrec`** work with nested scopes, and do not affect the meaning of global names
- Global names can be created with "define"
- Example:

```
(define hypt
    (lambda (a b)
       (sqrt (+ (* a a) (* b b)))))
(hypot 3 4)                    ➔ 5
```

Define: A “Global” **`let`**

```
(define fact
  (lambda (n)
     (if (<= n 1) 1
        (* n (fact (- n 1)))))
     )
  )
)

(fact 5)       ;  5! = 120
(fact 1000000); Wait for it, IT WILL WORK, TRY IT
```

Define: A "Global" **let**

User-defined list reversal:

```
(define myreverse
  (lambda (l)
    (if (null? l) l
        (append (myreverse (cdr l))
                (list (car l))
        )
    )
  )
)
```

```
(myreverse '(1 2 3)); (3 2 1)

(myreverse '(1 B $!@%1)); ($!@%1 b 1)

(equal? (myreverse '(1 2 3))
   (reverse '(1 2 3))); #t
```

A form of Iteration

- Try this:

```
(map + ‘( 1 2 3) ‘(10 20 30)) ;    (11 22 33)
```

- Pure functional languages don’t provide iterative constructs, but they do provide functions that can be applied to other functions over lists
- The above takes two lists, and applies ‘+’ to each corresponding pair of arguments: the first, the second the third
- Summing triples:

```
(map + '(1 2 3) '(10 20 30) '(100 200 300)); (111 222 333)
```

- Adding to products:

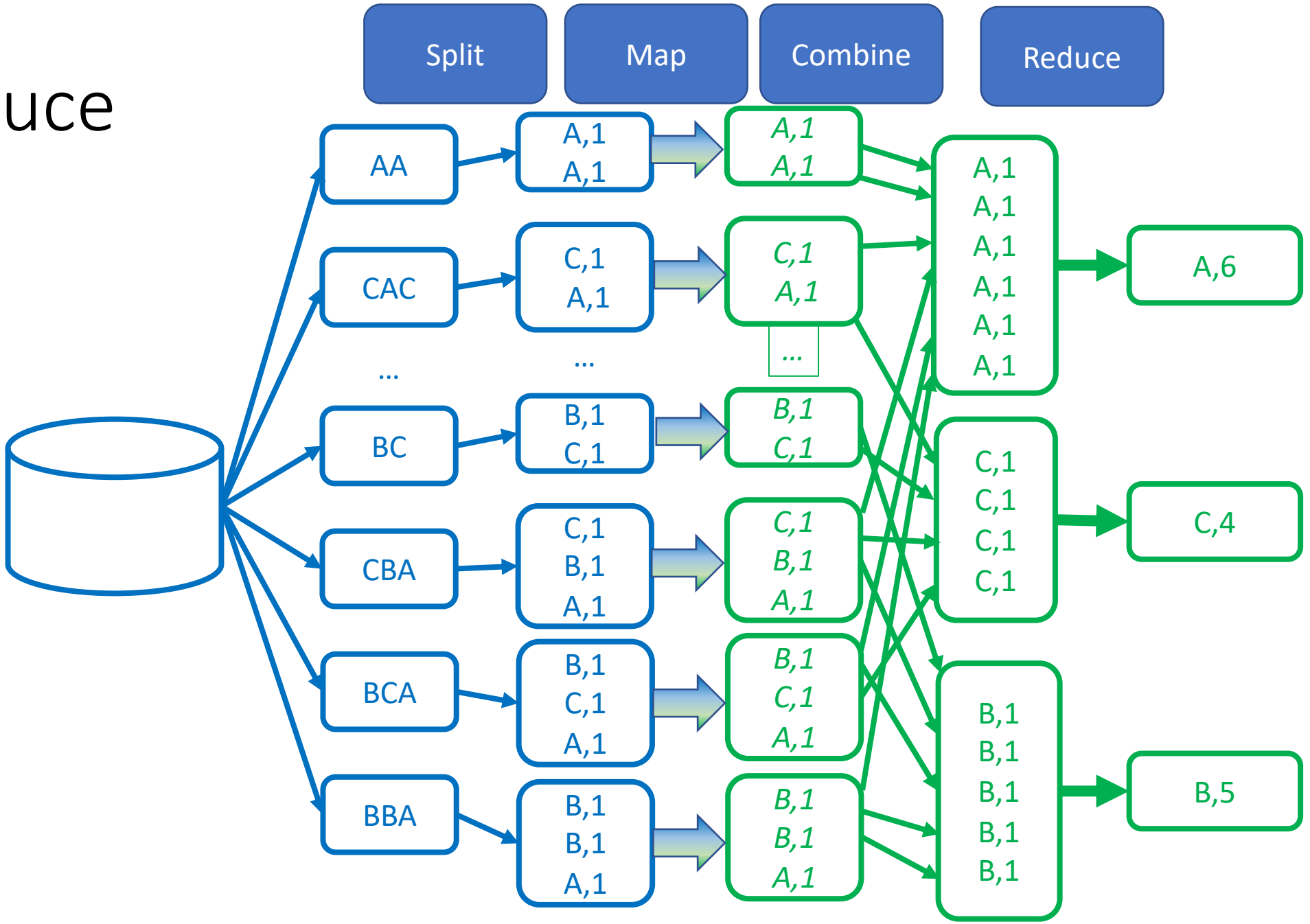
```
(map + '(1 2 3) (map * '(2 3 4) '(5 6 7)));    (11 20 31)
```

A form of Iteration

- Different Lisp dialects provide different forms of iteration
- Some deviate more (or less) from strict purely functional languages
- See:
  - do
  - for-each
  - cadr ← not a typo
  - caddddddddr ← as long as the list has sufficient elements
  - Many others, but dialect dependent

Map Reduce

- The Map-Reduce programming model (complex stuff) inspired in functional programming
- Provides impressive scalability in data centers
- Used on big-data applications
- Simple and elegant execution model for parallelism and distributed applications
- Many, many variations, e.g. Apache Hadoop
- Google, Facebook, etc, all have their own variant of Map-Reduce



B and *B* are different keys

C++ Lambdas

- Implementation of Lambdas in C++
- First introduced in C++11, and gradually updated in C++14 and C++17
- See more here
- Example:

```
#include <iostream>

auto make_function(int& x) {
  return [&]{ std::cout << x << '\n'; };
}

int main() {
  int i = 3;
  auto f = make_function(i);
  // the use of x in f binds directly to i
  i = 5;
  f(); // OK; prints 5
}
```

Install and try (MIT-)Scheme (if you want)

- From here: https://www.gnu.org/software/mit-scheme/
- In Mac OSX do: `brew install mit-scheme`
- Other install forms and instructions:

  `https://groups.csail.mit.edu/mac/users/gjs/6.945/dont-panic/#sec-2-2-3`
- In Ubuntu do: `sudo apt install scheme`
- Try one, anyone! just download it and try it
- A slightly longer tutorial can be found here
- Can also find a good online book here
- Read more:
  - Extents: can read more about it here
  - Garbage collection