Composing contracts An adventure in financial engineering

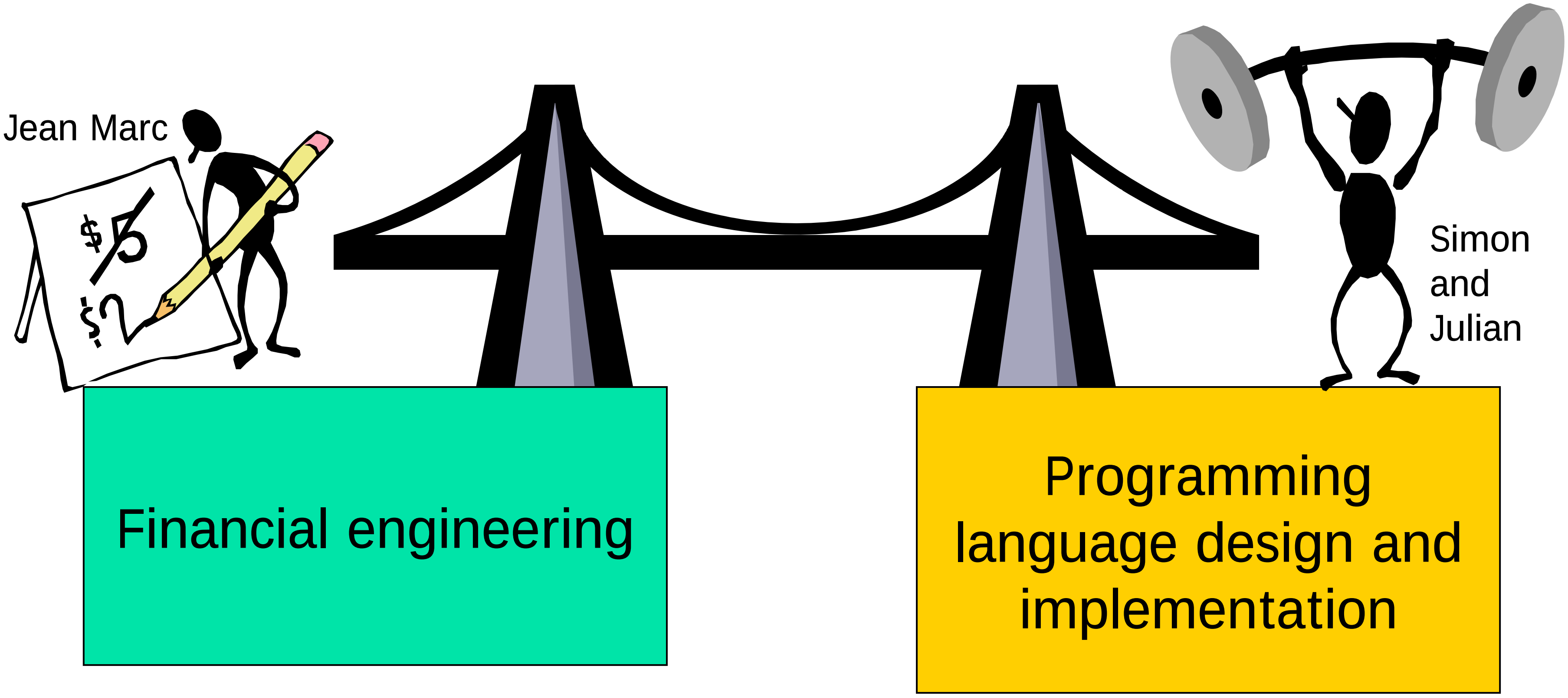
Simon Peyton Jones and Julian Seward,

Microsoft Research

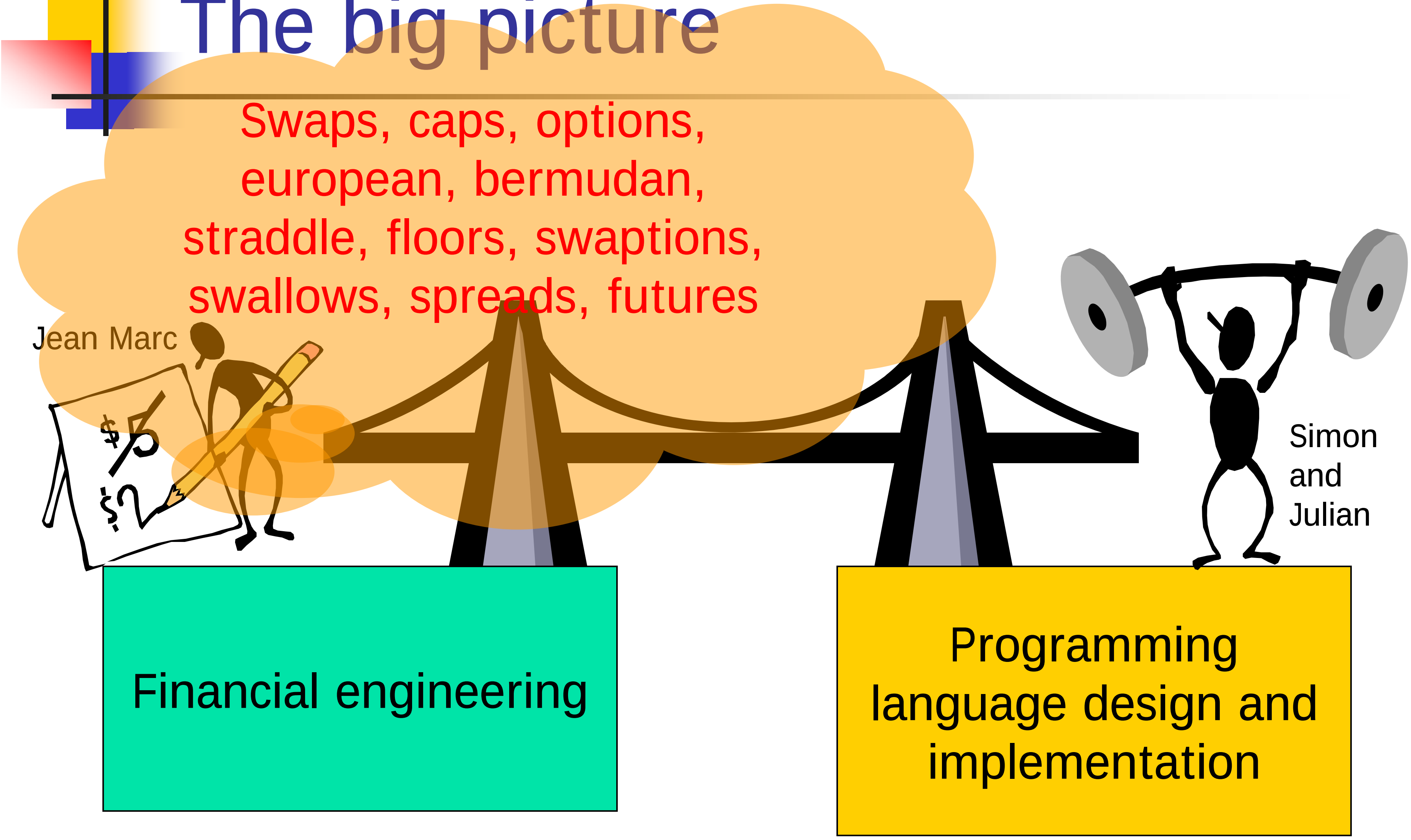
and

Jean Marc Eber, LexiFi Technologies

The big picture



The big picture



Combinator library

Denotational semantics

Operational semantics

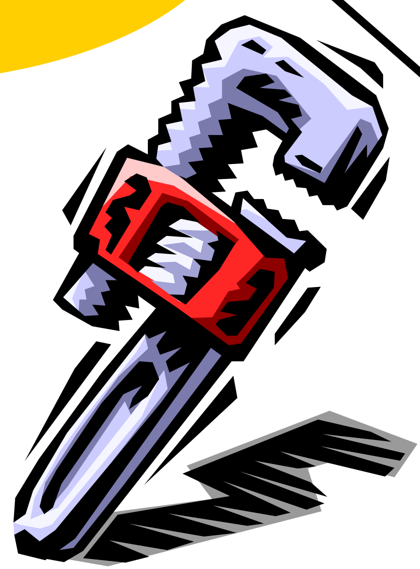
What we want to do

Precise description of a contract

Scheduling for back office

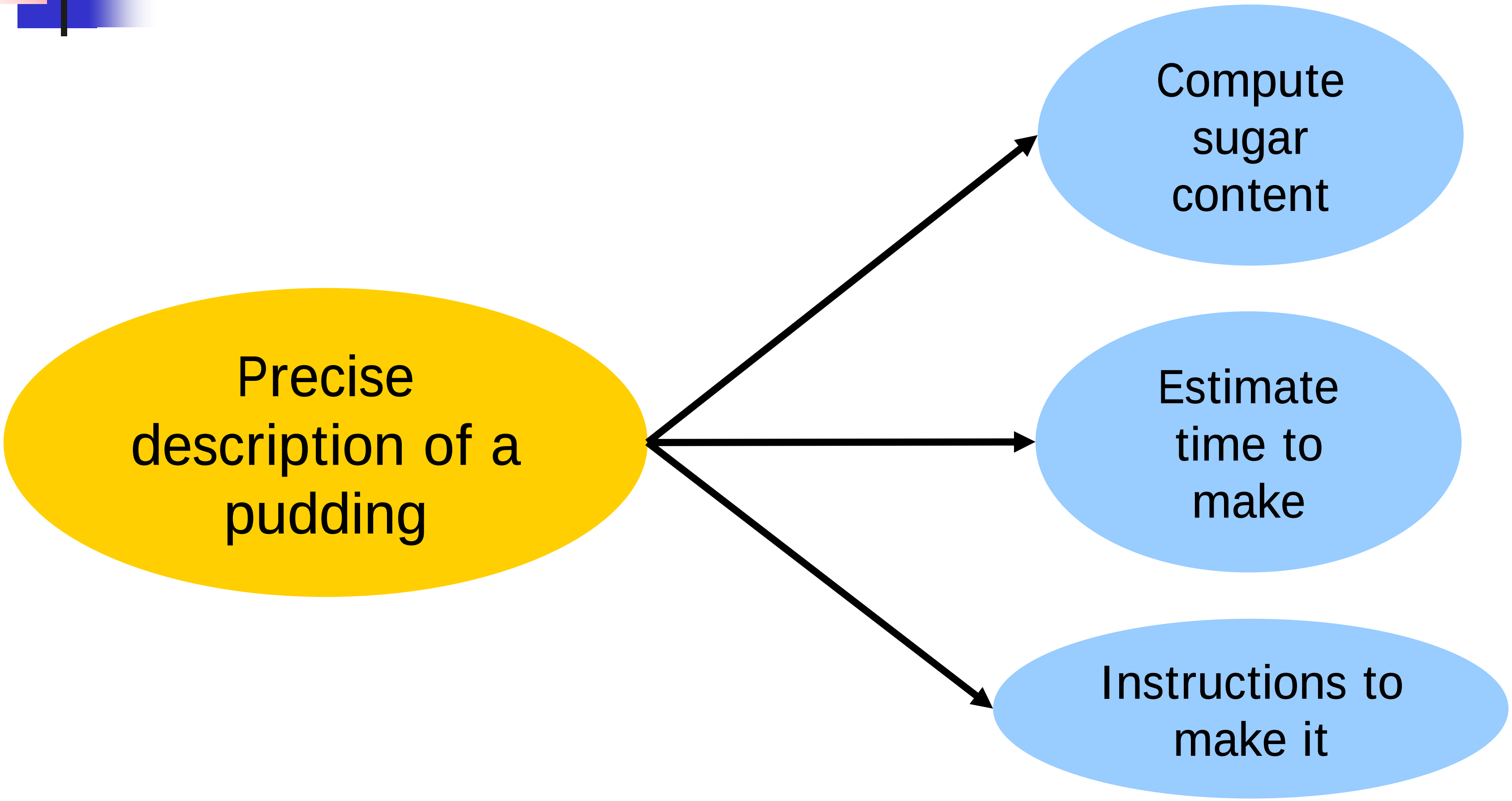
Valuation and hedging

Legal and documentation



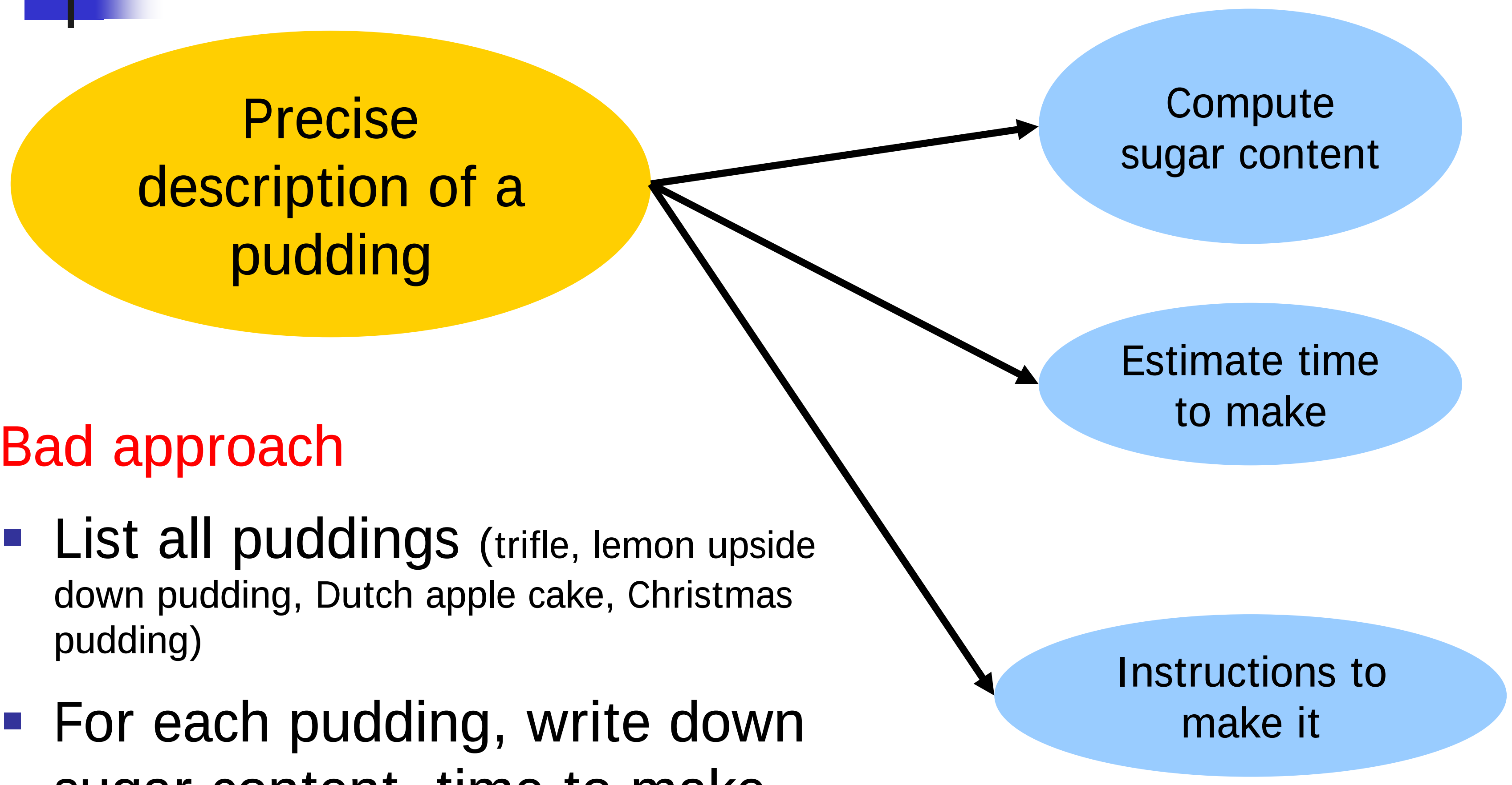
- etc....

What we want to do



- etc….

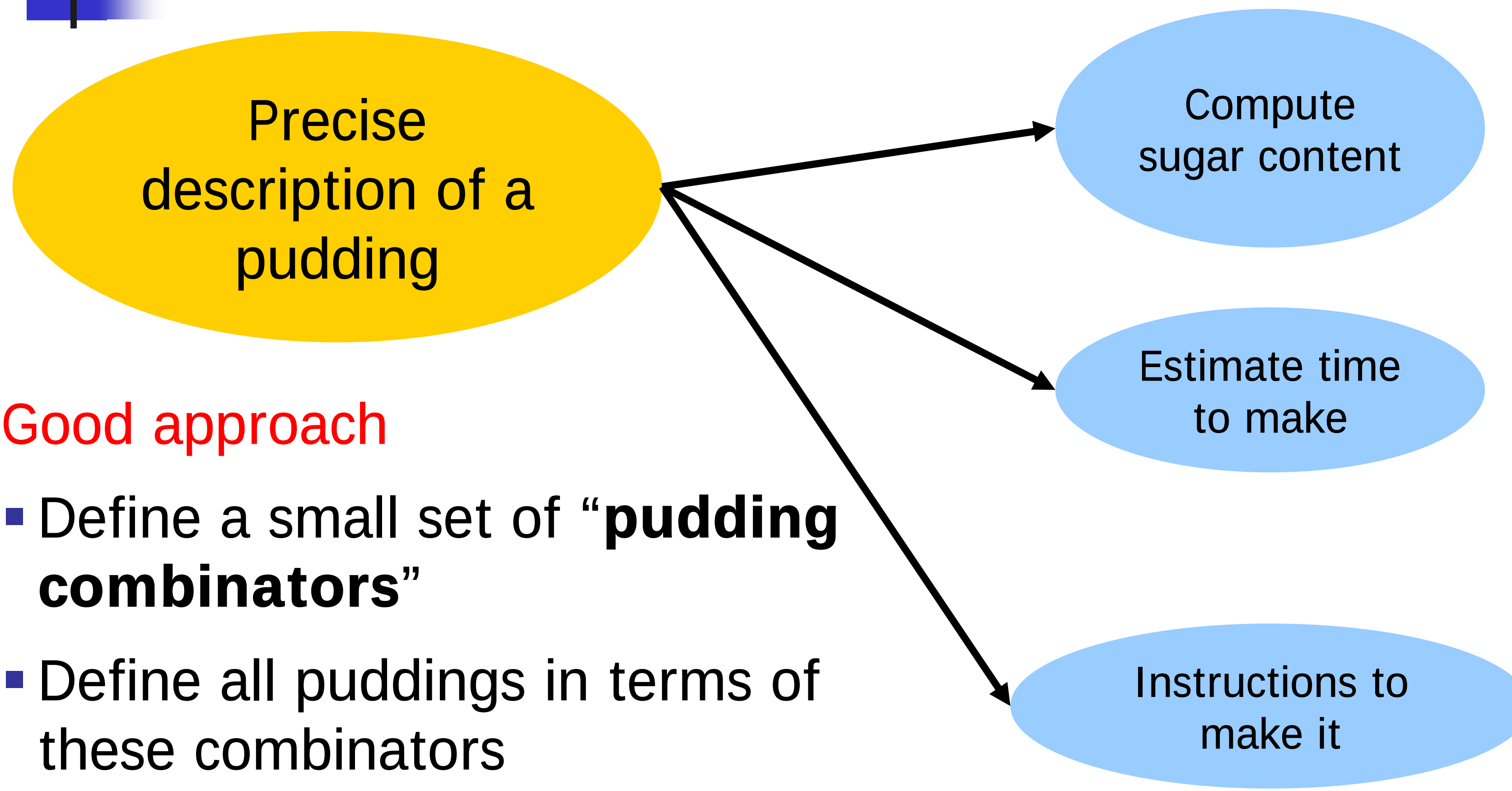
What we want to do



Bad approach

- List all puddings (trifle, lemon upside down pudding, Dutch apple cake, Christmas pudding)
- For each pudding, write down sugar content, time to make, instructions etc

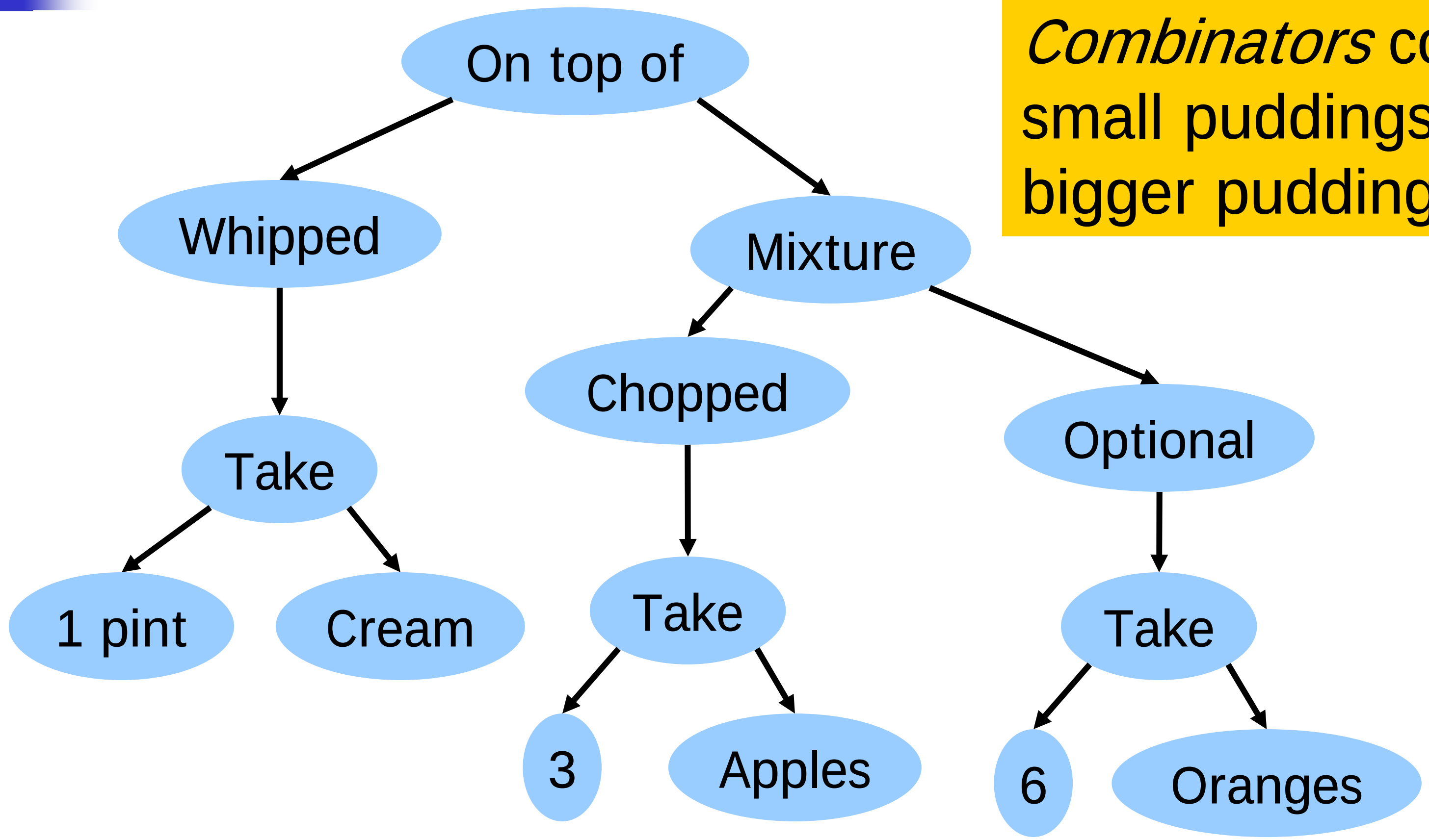
What we want to do



Good approach

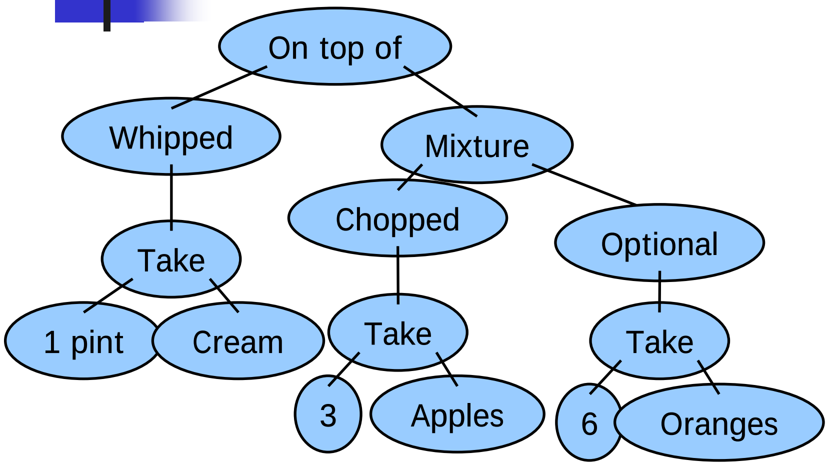
- Define a small set of “**pudding combinators**”
- Define all puddings in terms of these combinators
- Calculate sugar content from these combinators too

Creamy fruit salad



Combinators combine small puddings into bigger puddings

Trees can be written as text



Notation:

- **`parent child1 child2`**
- **`function arg1 arg2`**

```
salad       = onTopOf topping main_part
topping     = whipped (take pint cream)
main_part   = mixture apple_part orange_part
apple_part  = chopped (take 3 apple)
orange_part = optional (take 6 oranges)
```

Slogan: a domain-specific language for describing puddings

Building a simple contract

"Receive £100 on 1 Jan 2010"

```
c1 :: Contract
c1 = zcb (date "1 Jan 2010") 100 Pounds
```

```
zcb :: Date -> Float -> Currency -> Contract
-- Zero coupon bond
```

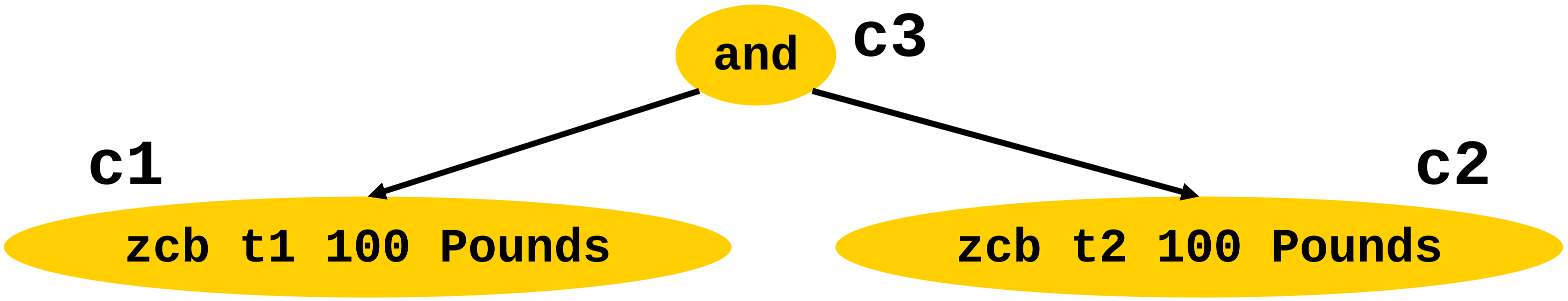
Combinators will appear in blue boxes

Combing contracts

```
c1,c2,c3 :: Contract
c1 = zcb (date “1 Jan 2010”) 100 Pounds
c2 = zcb (date “1 Jan 2011”) 100 Pounds

c3 = and c1 c2
```

```
and :: Contract -> Contract -> Contract
-- Both c1 and c2
```



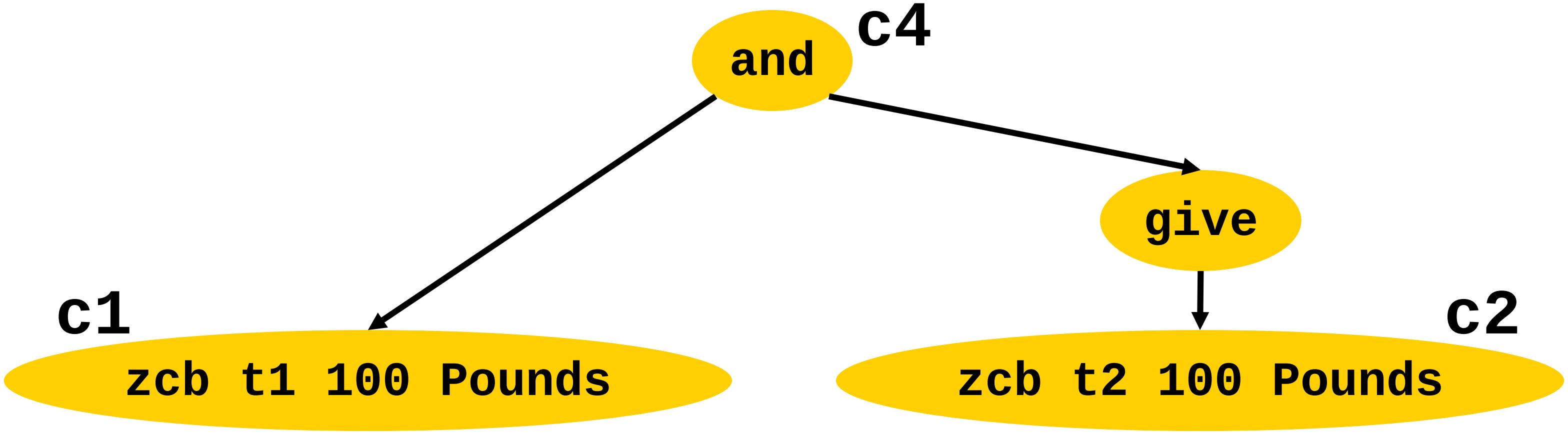
Inverting a contract

Backquotes for infix notation

```
c4 = c1 `and` give c2
```

```
give :: Contract -> Contract
-- Invert role of parties
```

- **and** is like addition
- **give** is like negation



New combinators from old

```
andGive :: Contract -> Contract -> Contract
andGive u1 u2 = u1 `and` give u2
```

- **`andGive`** is a new combinator, defined in terms of simpler combinators
- To the "user", **`andGive`** is no different to a primitive, built-in combinator
- This is the key to extensibility: **users can write their own libraries of combinators to extend the built-in ones**

Defining zcb

Indeed, **zcb** is not primitive:

```
zcb :: Date -> Float -> Currency -> Contract
zcb t f k = at t (scaleK f (one k))
```

```
one :: Currency -> Contract
-- Receive one unit of currency immediately

at :: Date -> Contract -> Contract
-- Acquire the contract at specified date

scaleK :: Float -> Contract -> Contract
-- Scale contract by specified factor
```

Acquisition dates

```
one :: Currency -> Contract
-- Receive one unit of currency immediately

at :: Date -> Contract -> Contract
-- Acquire the underlying contract at specified date
```

- If you acquire the contract **`(one k)`**, you receive one unit of currency k **immediately**
- If you acquire the contract **`(at t u)`** at time **`s`**<**`t`**, then you acquire the contract **`u`** at the (later) time **`t`**.
- If you acquire **`(at t u)`** later than **`t`**, you get nothing.

Observables

Pay me $1000 * (the number of inches of snow - 10) on 1 Jan 2002

```
c :: Contract
c = at "1 Jan 2002" (scale scale_factor (one Dollar))

scale_factor :: Obs Float
scale_factor = 1000 * (snow - 10)
```

```
scale :: Obs Float -> Contract -> Contract
-- Scale the contract by the value of the observable
-- at the moment of acquisition
```

Observables

An observable is an objectively-measurable, but perhaps time-varying quantity, or a value derived from such measurements

```
snow :: Obs Float
date :: Obs Date

const      :: a -> Obs a
(*), (-)   :: Obs Float -> Obs Float -> Obs Float
(>), (>=)  :: Obs a -> Obs a -> Obs Bool
```

```
scaleK k c = scale (const k) c
```

Acquisition triggers

Acquisition can be triggered by a boolean observable

```
c :: Contract
c = when late_snow (one GBP)

late_snow :: Obs Bool
late_snow = date > const “1 Apr 2003”  &&
            snow > 100
```

```
when :: Obs Bool -> Contract -> Contract
-- If you acquire (when o c), you acquire c at the
-- first moment when o subsequently becomes True
```

```
at t c  = when (date == const t) c
```

Choice

An **option** gives the flexibility to

- **Choose which** contract to acquire (or, as a special case, **whether** to acquire a contract)
- **Choose when** to acquire a contract (exercising the option = acquiring the underlying contract)

Choose **which**

- European option: at a particular date you may choose to acquire an "underlying" contract, or to decline

```
european :: Date -> Contract -> Contract
european t u = at t (u `or` zero)
```

```
or :: Contract -> Contract -> Contract
-- Acquire your choice of either c1 or c2
   immediately

zero :: Contract
-- A worthless contract
```

Reminder...

Remember that the underlying contract is arbitrary

```
c5 :: Contract
c5 =  european t1 (european t2 c1)
```

This is already beyond what current systems can handle

Choose **when**: American options

- The option to acquire 10 Microsoft shares, for $100, anytime between t1 and t2 years from now

```
anytime :: Obs Bool -> Contract -> Contract
-- Acquire the underlying contract at
-- any time the observable is True
```

anytime: Choose *when*

```
golden_handcuff = anytime (date >= t1 && date <= t2)
                          shares

shares = zero `or` (scaleK -100 (one Dollar) `and`
                    scaleK 10 (one MSShare))
```

or: Choose *whether*

MS shares are a “currency”

Summary so far

- Only 10 combinators (after many, many design iterations)
- Each combinator does one thing
- Can be combined to describe a rich variety of contracts
- Surprisingly elegant

But what does it all mean?

- We need an absolutely precise specification of what the combinators mean: their semantics
- And we would like to do something useful with our (now precisely described) contracts
- One very useful thing is to compute a contract's value

Use denotational semantics

- The **denotation** of a program is a mathematical value that embodies what the program "means"
- Two programs are **equivalent** if they have the same denotation
- A denotational semantics should be **compositional**: the denotation of (P1 + P2) is gotten by combining somehow the denotations of P1 and P2

Processing puddings

- Wanted: S(P), the sugar content of pudding P

S(**`onTopOf p1 p2`**)	= S(**`p1`**) + S(**`p2`**)
S(**`whipped p`**)	= S(**`p`**)
S(**`take q i`**)	= q * S(i)
...etc...	

- When we define a new recipe, we can calculate its sugar content with no further work
- Only if we add new combinators or new ingredients do we need to enhance S

Processing puddings

- Wanted: S(P), the sugar content of pudding P

S(**onTopOf p1 p2**)	= S(**p1**) + S(**p2**)
S(**whipped p**)	= S(**p**)
S(take q i)	= q * S(i)
...etc...	

S is ***compositional***

To compute S for a compound pudding,

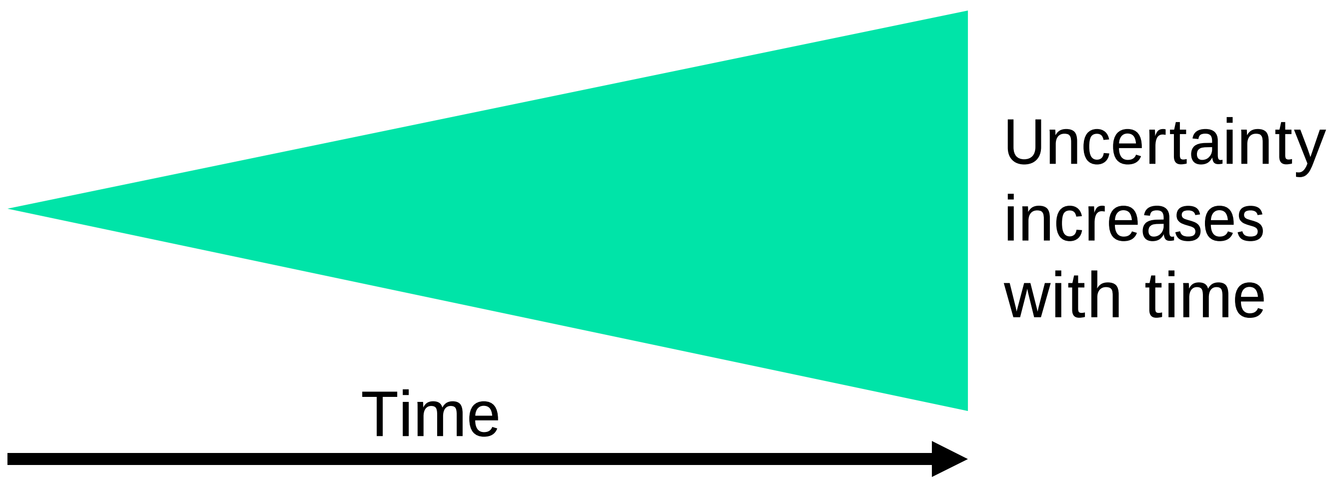
- Compute S for the sub-puddings
- Combine the results in some combinator-dependent way

What is the denotation of a contract?

Main idea: the denotation of a contract is a **random process** that models the value of acquiring the contract at that moment.

$\mathcal{E}$: Contract -> RandomProcess

RandomProcess = Time -> RandomVariable



Compositional valuation

Add random processes point-wise

$\mathcal{E}$(`c1 `and` c2`) = $\mathcal{E}$(`c1`) + $\mathcal{E}$(`c2`)

$\mathcal{E}$(`c1 `or` c2`) = max($\mathcal{E}$(`c1`), $\mathcal{E}$(`c2`))

$\mathcal{E}$(`give c`) = - $\mathcal{E}$(`c`)

$\mathcal{E}$(`when o c`) = discount($\mathcal{E}$(`o`), $\mathcal{E}$(`c`))

$\mathcal{E}$(`anytime o c`) = snell($\mathcal{E}$(`o`), $\mathcal{E}$(`c`))

...etc...

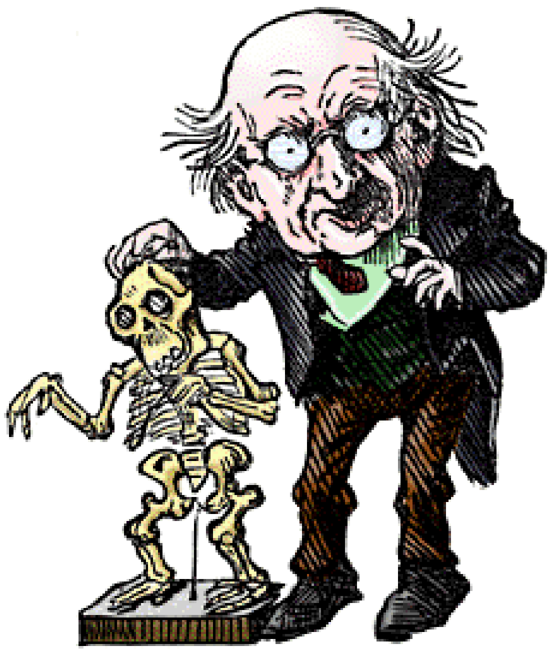
Standard financial operators

This is a **major payoff**! Deal with the 10-ish combinators, and we are done with valuation!

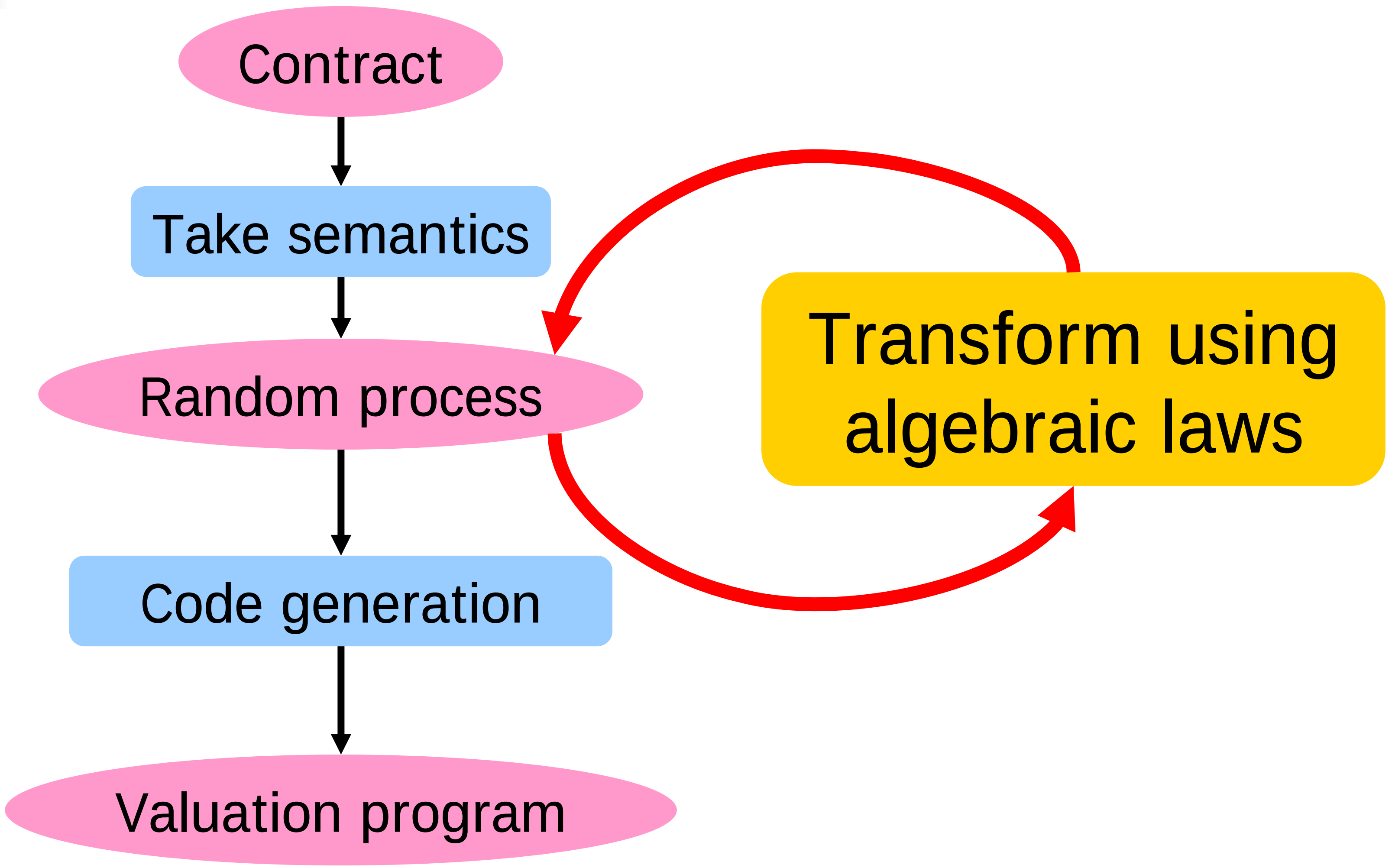
Reasoning about equivalence

- Using this semantics we can prove (for example) that
  `anytime o (anytime o c) = anytime o c`

- Depends on algebra of random processes (snell, discount, etc). Bring on the mathematicians!

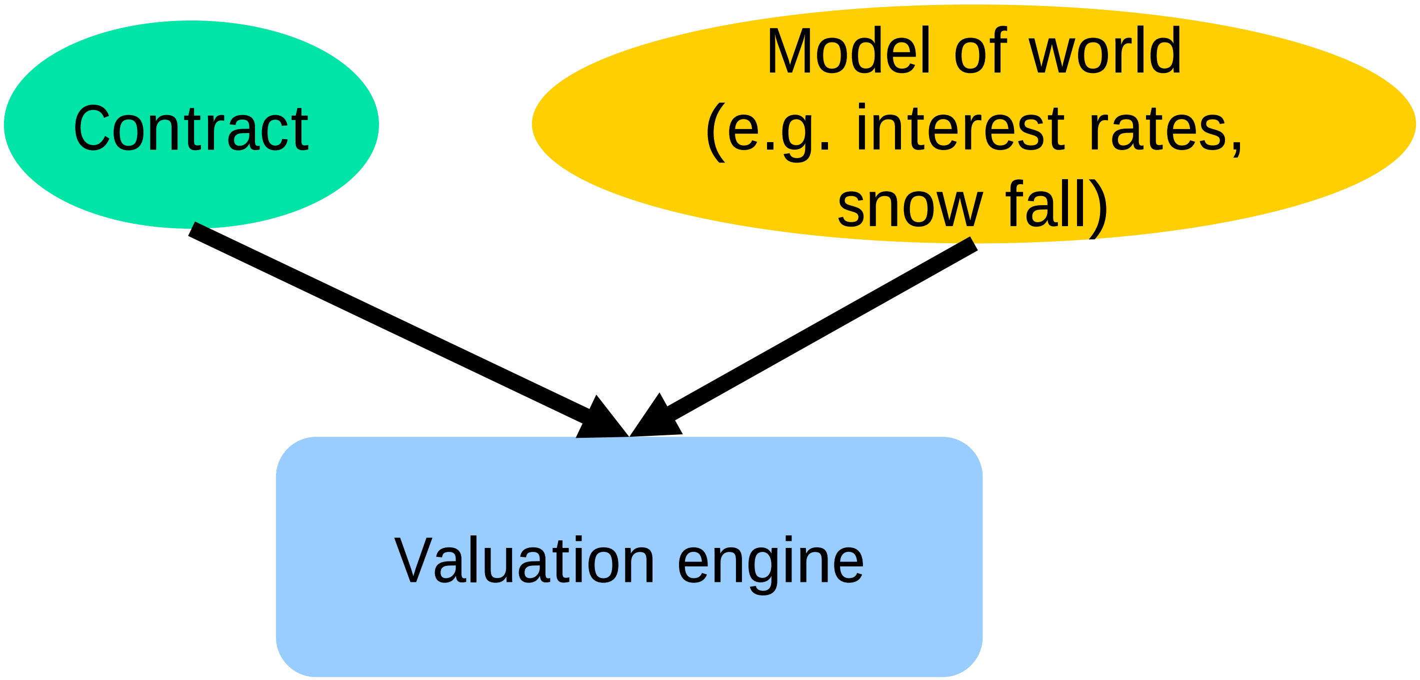


A compiler for contracts

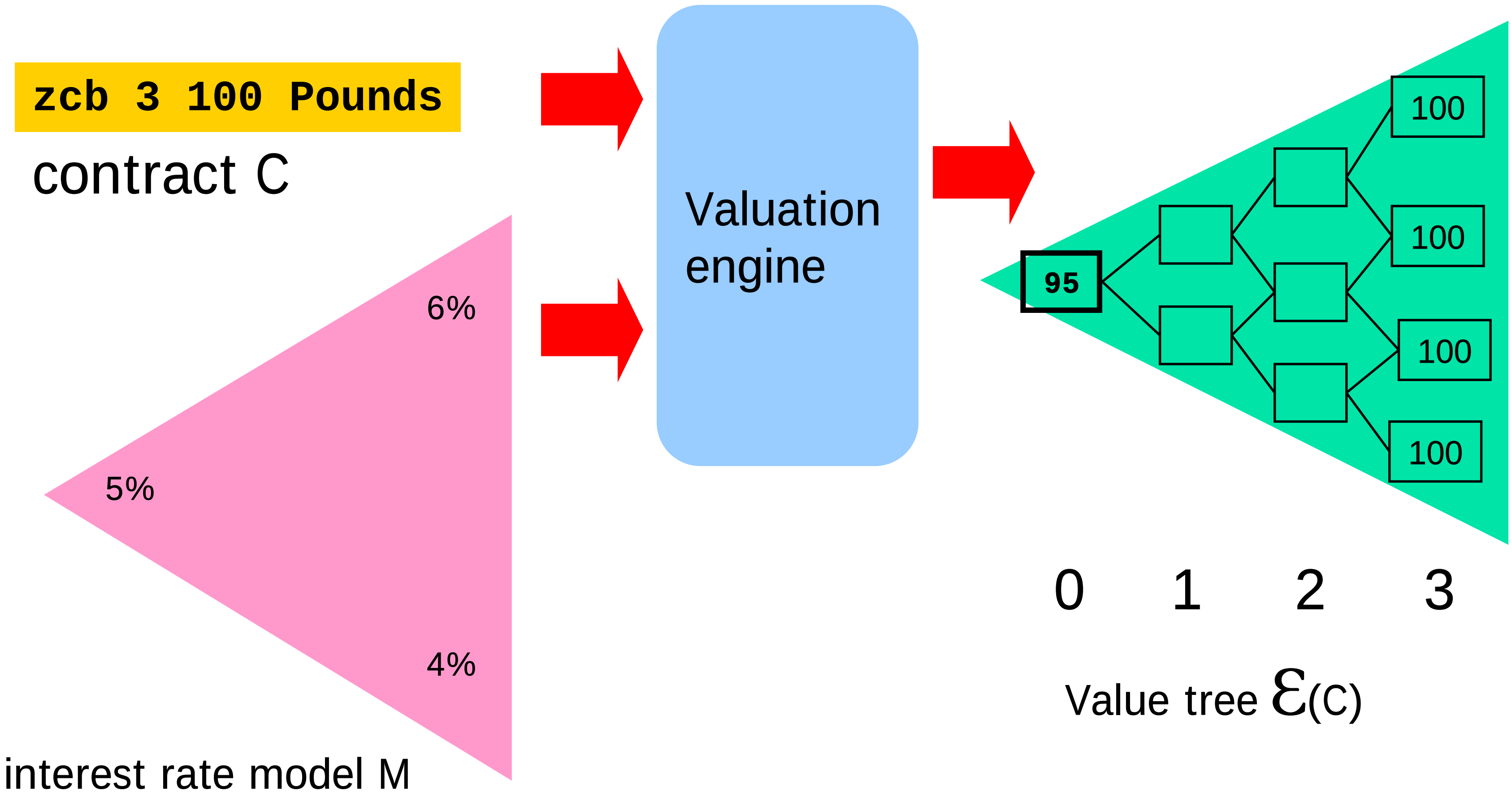


Valuation

- There are many numerical methods to compute discrete approximations to random processes (*tons and tons and tons and tons and tons and tons and tons and tons and tons and tons and tons and tons and tons* of existing work)

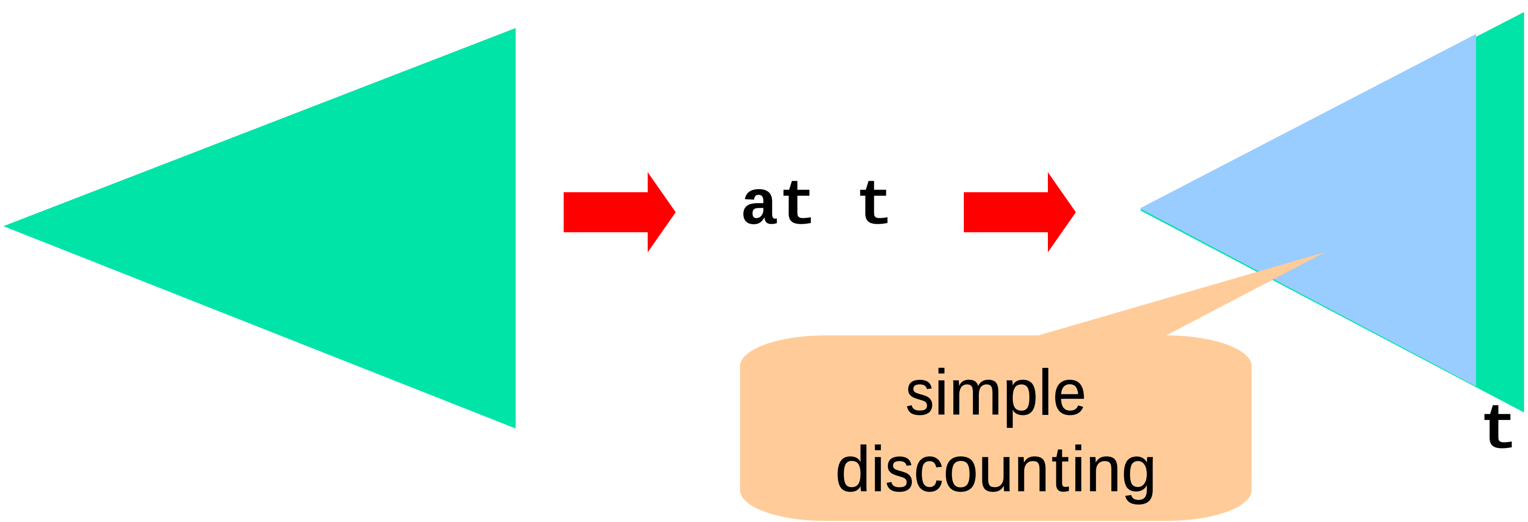


One possible evaluation model: BDT



Space and time

- Obvious implementation computes the value tree for each sub-contract
- But these value trees can get BIG
- And often, parts of them are not needed



Haskell to the rescue

"Lazy evaluation" means that

- data structures are computed incrementally, as they are needed (so the trees never exist in memory all at once)
- parts that are never needed are never computed

Slogan

We think of the tree as a first class value "all at once"

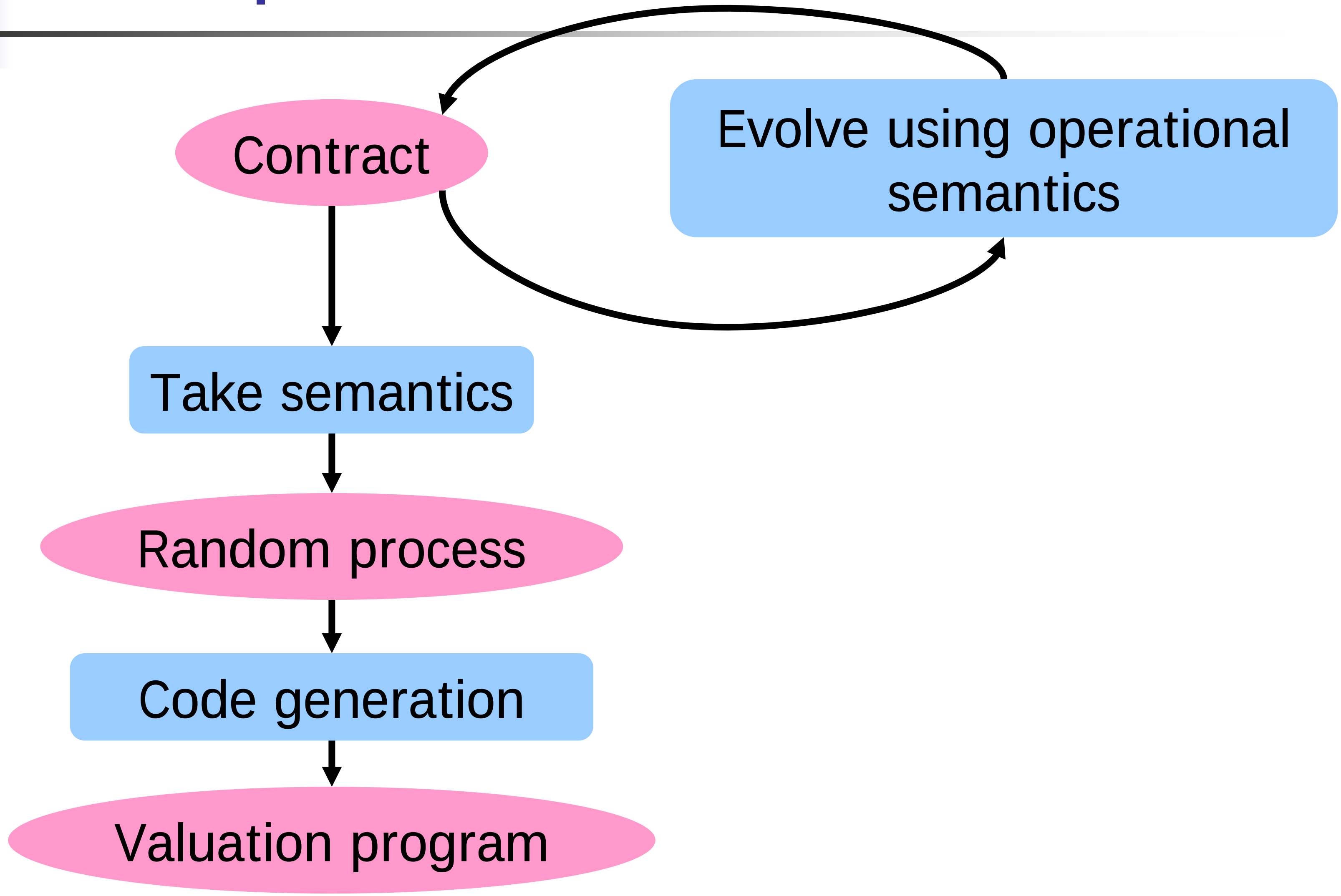
but it is only materialised "piecemeal"

An **operational** semantics

- As time goes on, a contract evolves

  e.g. `zcb t1 n k `and` zcb t2 n k`
- Want to value your current contract "book"
- So we want to say formally how a contract, or group of contract **evolves with time**, and how it **interacts with its environment** (e.g. emit cash, make choice)
- Work on the operational semantics of programming languages is directly relevant (e.g. bisimulation)

A compiler for contracts



Beyond financial contracts

[Henglein FLACOS 2007]

Section 1. The attorney shall provide, on a non-exclusive basis, legal services up to (n) hours per month, and furthermore provide services in excess of (n) hours upon agreement.

Section 2. In consideration hereof, the company shall pay a MDCC 16 monthly fee of (amount in dollars) before the 8th day of the following month and (rate) per hour for any services in excess of (n) hours 40 days after the receipt of an invoice.

Section 3. This contract is valid 1/1-12/31, 2008.

Again: a domain specific language

```
letrec
extra (att, com, invoice, pay) =
    ( Success
    + transmit (att, com, invoice, T2).
      transmit (com, att, pay, T3 | T3 <= T2 + 45d))

legal (att, com, fee, invoice, pay, n, m, end) =
     transmit (att, com, H, T | n < T and T <= m).
         (    extra (att, com, invoice, pay)
         ||   transmit (com att, fee, T | T <= m + 8d)
         ||   ( legal (att, com, fee, invoice, pay, m, min(m + 30d,end), end)
                + transmit (att, com, end, T | end <= T)))
in
  legal ("Attorney","Company",10000,invoice,pay,0,30,360)
```

Summary

Routine for us, radical stuff for financial engineers

- A small set of built-in combinators: named and tamed
- A user-extensible library defines the zoo of contracts
- Compositional denotational semantics, leads directly to modular valuation algorithms
- Risk Magazine Software Product of the Year Prize
- Jean-Marc has started a company, LexiFi, to commercialise the ideas. Paying customers, typesafe .NET interoperation, sophisticated pricing models etc.

Summary

- A small set of built-in combinators: named and tamed
- A user-extensible library defines the zoo of contracts
- Compositional denotational semantics, leads directly to modular valuation algorithms
- Risk Magazine Software Product of the Year Prize
- Jean-Marc has started a company, LexiFi, to commercialise the ideas
- **Beats higher order logic hands down for party conversation**