Chapter 1: Building Abstractions with Procedures

Foreword

- a computer program needs to be performant - need to be expandable and connectible with other programs (modular)
- "continual unfolding" of the model
- as a program gets large, its proof of logic and correctness gets more complicated
    - grow large programs from small ones which are consistent and correct (idioms) and connect them.
    - organize interfaces
- know algorithms and idioms to optimize programming/performance

Chapter 1 - Building Abstraction with Procedures

“The acts of the mind, wherein it exerts its power over simple ideas, are chiefly these three: 1. Combining several simple ideas into one compound one, and thus all complex ideas are made. 2. The second is bringing two ideas, whether simple or complex, together, and setting them by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of relations. 3. The third is separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all its general ideas are made.” John Locke

> A great Quote to demonstrate how we understand and abstract ideas

Computational process:

- processes manipulate data (driven by a program)

- composed by symbolic expressions
- executes programs

1.1 The Elements of Programming

Programming language serves as a framework where we organize ideas and form from simple concepts more complex ones:

primitive expressions → means of combination → means of abstraction

- procedures: description of rules to manipulate data
- data: information that is manipulated

1.1.1 Expressions

Expressions are *evaluated* by an *interpreter*. More complex expressions are combinations from fundamental expressions. operands are connected via operators.

Nesting: Connections and inception of suboperations.

1.1.2 Naming and the Environment

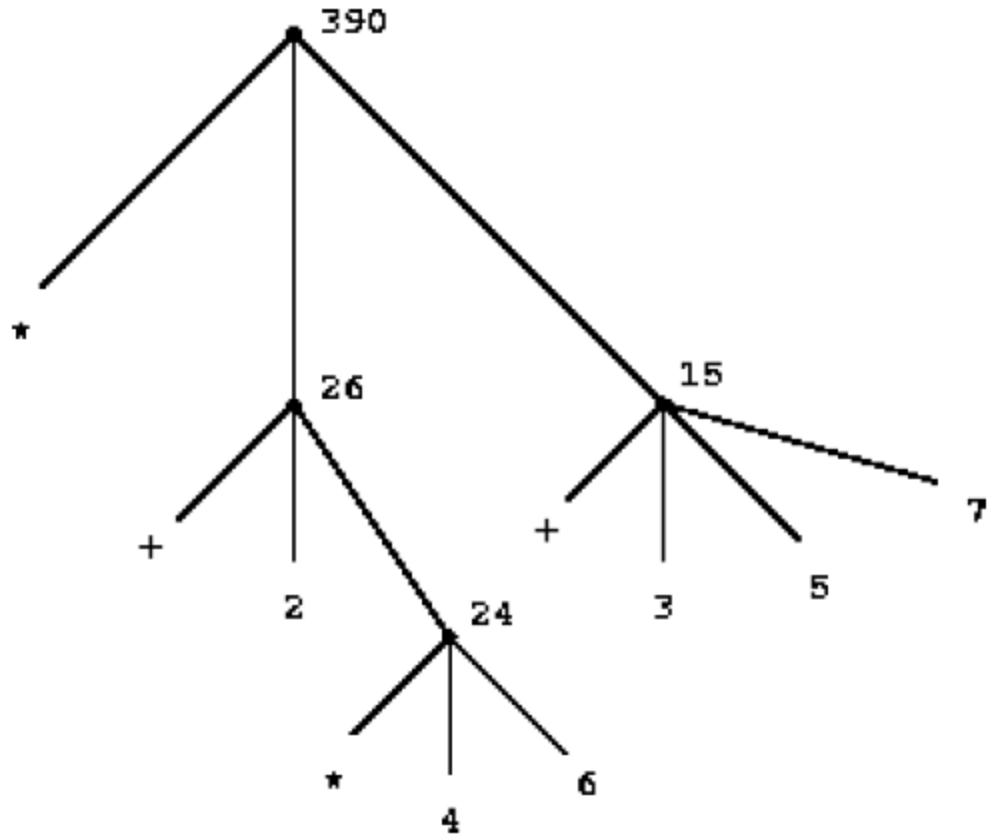
Name identifies a *variable* whose *value* is the object. Defines are the simplest means of abstraction. f.e. define pi 3.14, define r 1, define circumference (2*pi*r)

1.1.3 Evaluating Combinations

1. Evaluate the subexpressions of the combination
2. Apply the procedures with the operators to the operands of the subexpressions

In nature these steps are of course recursive, when the expressions are not elementary.

Example: Tree accumulation of simple math operations



- Numbers and arithmetic operators are fundamental primitive data and procedures
- Nesting to combine operations to more complex forms

In SICP LISP is used as a programming language.

Example: compound procedure

```
(define (square  x)          (*          x     x))
  ↑        ↑     ↑             ↑          ↑     ↑
 To      square something, multiply   it by itself.
```

General form of compound procedures:

```
(define (<name> <formal parameters>) <body>)
```

Out of the square define, one can now combine it with a summation or similiar constructs

```
(define (sumofsquares x y) (+ (square x) (square y)))
```

1.1.5 The Substitution Model for Procedure Application

Going through the following example, the substitution model is shown

```
(define (f a b)  (sumofsquares (* a a) (+ a b) ) //a^4 + (a+b)^2

(f  3 5)
(sumofsquares (* 3 3) (+ 3 5))
( + (* 9 9) (* 8 8 ))
(+ 81 72)
153
```

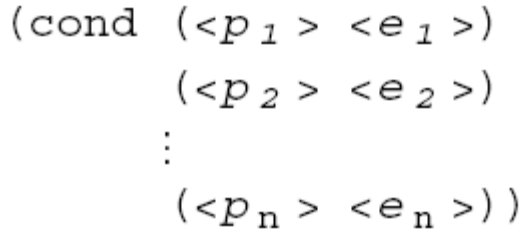
- this is not how the interpreters work, it's more a way to understand and think about procedure application

Applicative order versus normal order

It's also possible to reduce the evaluation to primitive operands and then apply the operators. It first fully expands and then calulates (normal order evaluation). Evaluate the arguments and then apply is the method that the interpreter uses (applicative order evaluation)

1.1.6 Conditional Expressions and Predicates

Defines are mighty, but we still need conditional expressions to test different cases



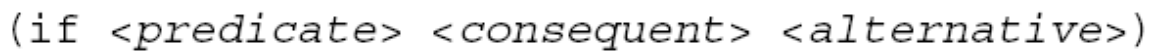
For example

```
(define (greaterzero x) (cond (  (> x 0) 1)((< x 0) 0)(= x 0) 0) ) )
```

The pairs (<p> <e>) are called clauses, <p> is a predicate, either false or true. <e> is the *consequent expression*, which happens when <p> is true.

It's also possible to define **else** statements or if's

```
(define (abs x) (cond ((<x 0) (-x))( else x)))
(define (abs x) (if (< x 0)  (-x)  x)
```



Logical boolean predicates:

```
(and <e1> ... <en>)
```

```
(or <e1> ... <en>)
```

```
(not <e>)
```

For example: **x^2 < 5x < x^3**

```
(and( ( < (square x) (* x 5) ) ( < (* x 5) (* (square x) x) ) )
```

One may also define Operators as the following

```
(define (>= x y) (or (x > y) (= x y) ) )
(define (>= x y) (not (< x y) ) )
```

Exercise 1.5. Ben Bitdiddle has invented a test to determine whether the interpreter he is faced with is using applicative-order evaluation or normal-order evaluation. He defines the following two procedures:

```
(define (p) (p))
(define (test x y)
  (if (= x 0)
      0
      y))
```

Then he evaluates the expression

```
(test 0 (p))
```

What behavior will Ben observe with an interpreter that uses applicative-order evaluation? What behavior will he observe with an interpreter that uses normal-order evaluation? Explain your answer. (Assume that the evaluation rule for the special form `if` is the same whether the interpreter is using normal or applicative order: The predicate expression is evaluated first, and the result determines whether to evaluate the consequent or the alternative expression.)

1. applicative order evaluation

```
(define (p) (p))
(define (test x y) (if (=x 0) 0 y) )

(test 0 (p))
(test 0 p)
(if( =0 0) 0 p))
(if(true) 0 p))
(0)
```

2. normal order evaluation

```
(define (p) (p))
(define (test x y) (if (=x 0) 0 y) )

(test 0 (p))
((if(=0 0) 0 (p))
(if(true) 0 (p))
(0)
```

We can see here, that the (p) expression didn't need to be evaluated. This may save time in this specific scenario.

Local Names

procedures should be abstracted with local variables - independency of parameter names is wished. With this, subroutines may use the local variable as input, but are in fact not dependent on them.

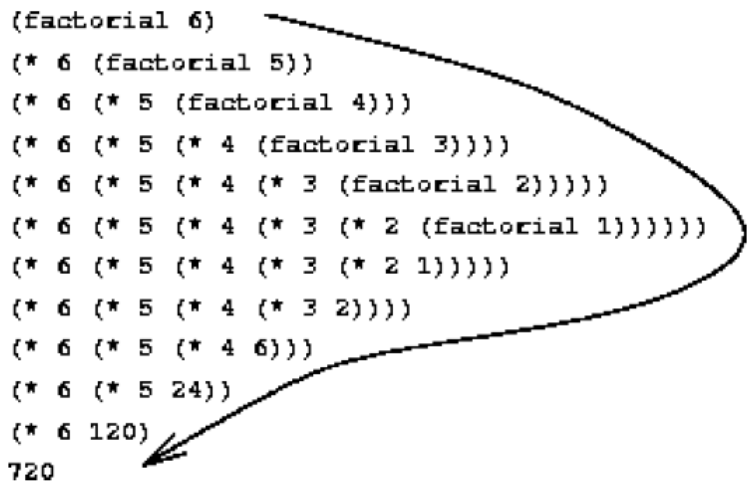
Nesting with definitions is called *block structure*. Calling subroutines with local parameters is called *lexical scoping*. It breaks large programs into tractable pieces, which are only connected via the parameters.

1.2 Procedures and the Processes They Generate

Procedures are patterns for the *local evolution* of the computations. Specifying global evolutions is far more difficult - but it's possible to connect different scenarios and make a great picture out of the small puzzle pieces.

1.2.1 Linear Recursion and Iteration

This applies to f.e. just the recursive function until it's all elementary expressions and then computes them backwards.



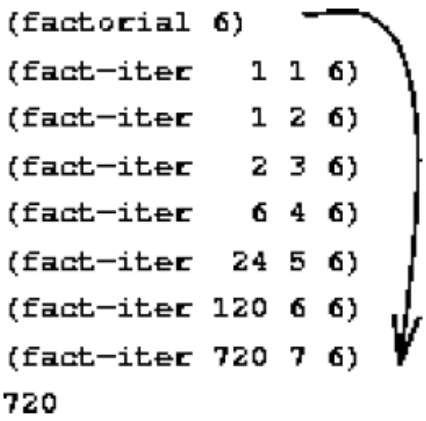
The Factorial function can be easily written in Lisp as

```
(define (factorial n) (= n 1) 1 (* n factorial(- n 1)))
```

One may also calculate this from the other side, starting from the lowest value going up.

```
(define (product start end) ( if (< start end)(* start product start+1 end) end )
(define (factorial n) (* 1 product 1 n) )
```

Those two algorithms look pretty much the same, but are indeed very different. The first one builds up a chain, called *deferred operations* and is recursive. The other one is a *linear recursive process* as the number of steps grows linearly.



These are **iterative** processes. which are characterized by a fixed amount of states, which are updated when going to the next step by a given rule. The iterative step is way more practical than the recursive, as the recursive has to be completed fully but the iterated can be just completed where it stopped. The recursive process needs way more memory than the iterated one.

An iterative implementation of the 1 up to n factorial might be as following:

```
(define (iter-product value cnt stop))
if(< start stop) (iter-product ((* value cnt) (+ cnt 1) stop))) value)
(define (factorial n) (iter-product 1 1 n) )
```

> Keep in mind: **procedure ≠ process** The procedure just refers to the fact, that it refers to itself instead of the process itself.

tail recursive implementations can iterate with an ordniary procedure call mechanism - it has a constant space, not one which is expanding by step. We just need the current state and we can continue with this configuration.

1.2.2 Tree Recursion

A classical example for *tree recursion* is the fibonacci sequence, defined as

$$F_0 = 0,\, F_1 = 1,\, F_n = F_{n-1} + F_{n-2}, \quad n \geq 2$$

Translating it into Lisp leads to

```
(define f(n) (cond(> n 1) (+ F (- n 1) F(- n 2))(= n 1) 1 (=n 0) 0) )
```

It's called tree recursion because of the branched structure when writing down each fibonacci sequence (which needs two "older" ones)

It's computationally really bad, when doing it like above, because of many many duplications.

It grows like

$$\mathcal{O}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

which is huge for big n.

Of course it's also possible to formulate the Fibonacci problem via an iterative process. Let's prove a simple thing first and then derive an iterative process to calculate the Fibonacci sequence.

$$a \leftarrow a + b$$
$$b \leftarrow a$$
$$\text{init} : F_0 = b,\ F_1 = a$$

Looking at this transformation it has the structure of something like

$$x_{n+1} \leftarrow x_n + x_{n-1}$$
$$x_n \leftarrow x_{n-1}$$

This can be done, as we identify

$$a_n := x_n, b_n = x_{n-1}$$

since the old a always gets mapped to the new b, causing the shift in index.

We can just shift the index in the above equation and get

$$x_n \leftarrow x_{n-1} + x_{n-2}$$

Which is exactly the Fibonacci sequence. So... What did we win here? We got a Transfomation of **CURRENT STATE VARIABLES** which we can now apply to an iterative process as following:

```
(define (fibonacci n) (iter 1 0 n))
(define (iter a b n) (if(<= a n) (iter (+ a b) (a) (- n 1))) a ))
```

This will be a linear iteration and not exponentially growing. This example showed, that it makes sense to think about smart ways to make iterations

instead of recursions! Smart compilers may transform tree-recursive problems into more efficient processes.

1.2.3 Orders of Growth

The order of growth yields a gross measure for the order of resources needed to complete the algorithm with size n. The size is a mere abstract definition, it can be different things thus it needs to be declared upfront when saying it grows like a^n.

We say R(n) has order of growth O(f(n)) if there exist two positive constants k and l independent of n with

$$k \cdot f(n) \leq R(n) \leq l \cdot f(n)$$

As an example, the recursive factorial requires O(n^1) steps and O(n^1) space, while the iterative requires also O(n^1) steps, but a constant O(n^0) space.

Logarithmic growth is a special interesting case, which will be investigated further in the following. Here, doubling the size increases the resource by a constant. Thus, algorithms with logarithmic order of growth are great and in general desired for complex algorithms. A classic example for this is the primes check. (Primes are in P, a famous paper finding an algorithm to check if number a is a prime, with logarithmic growth!)

1.2.4 Exponentiation

When computing an exponential of a given number, you have a base b with exponent n. Computing it recursively like following is quite straight-forward

$$b^n = b \cdot b^{n-1}$$

```
(define (exp b n) (if( = n 0) 1) (* exp n (n-1)) ) )
```

One can of course define an iterative process here too. It's also possible to successive square like in the following:

$$b^2 = b \cdot b$$
$$b^4 = b^2 \cdot b^2$$
$$...$$

Of course this only works with exponents with a power of two, but we can generalize it to

$$b^n = (b^{\frac{n}{2}})^2, \quad n\ even$$
$$b^n = b \cdot b^{n-1}, \quad n\ odd$$

This grows logarithmically with n in both space and number of steps! Awesome! F.e. n=1000 only takes 14 multiplications with the faster method. Always try to get logarithmic growths, for large computations it's god sent. When doing really smart things, it's also possible to calculate the Fibonacci sequence with logarithmic steps.

1.2.5 Greatest Common divisor

The greatest common divisor can be really messy to compute if you don't know some smart algorithms. Of course there exist some smart people who saw that: GCD(a,b) = GCD(b,r), when r is defined as the remainder if you divide a by b! Think about it, you can recursively apply this! GCD(b,r) = GCD(r, r2) = ... With this you always get smaller numbers. This is called the Euclid Algorithm (yes it's really old)

```
(define (gcd a b) if(= b 0) a (gcd b (remainder a b) ) )
```

1.3 Formulating Abstractions with Higher-Order Procedures

1.3.1 Procedures as arguments

As we don't want to hardcode expressions like

```
(* 3 3 3)
```

we define procedures like

```
(define (cube x) (* x x x) )
(cube 3)
```

For realistic non-primitive programs, formulating everything without defines makes no sense.

Also: How about defining procedures that take procedures as arguments? This will further abstract the program and make it easier to handle, and also to test!

We can for example define a sum of terms (like cubes or fractions, ...)

$$\sum_{n=a}^{b} f(n)$$

```
(define (sum term a next b)
  (if (> a b)
    0
    (+ (term a) (sum term (next a) next b))))
```

Incrementing always by 1 for example next would of course be defined as

```
(define (next n) (+ n 1))
```

Summing up all the cubes in a closed interval then looks like

```
(define (sumcubes a b )
  (sum cube a next b) )
```

Here we applied abstraction with one more layer than in the beginning.

Going a step further, we can define for example an integral with our sum

```
(define (integral f a b dx)
(define (add-dx x) (+ x dx))
(* (sum f (+ a (/ dx 2.0)) add-dx b) dx))
```

1.3.2 Constructing Procedures Using Lambda

A great way to save some defines, we can use lambdas, which creates procedures locally.

```
(define (integral f a b dx)
(* (sum f (+ a (/ dx 2.0)) (lambda (x) (+ x dx)) b) dx))
```

☝ *(lambda (<formal-parameters>) <body>)*

Lisp has the possibility to create local variables in a define like:

```
(define (f x y)
(let ((a (+ 1 (* x y)))
(b (- 1 y)))
(+ (* x (square a))
(* y b)
(* a b))))
```

Let expressions are basically the same like lambdas, but only for local variables with easier syntax.