

Chapter 1

Woo Jia Hao

May 15, 2020

Contents

1	The Elements of Programming	2
1.1	Expressions	2
1.2	Naming and the Environment	4
1.3	Evaluating Combinations	5
1.4	Compound Procedures	7
1.5	The Substitution Model for Procedure Application	8
1.5.1	Applicative order vs normal order	9
1.6	Conditional Expressions and Predicates	10
1.6.1	Logical composition operators	12
1.7	Example: Square Roots by Newton's Method	13
1.7.1	Newton's method of successive approximation	13
1.8	Procedures as Black-Box Abstractions	14
1.8.1	Local names	14
1.8.2	Internal definitions and block structure	15
2	Procedures and the Processes They Generate	16
2.1	Linear Recursion and Iteration	16
2.1.1	Linear recursive process	19
2.1.2	Linear iterative process	19
2.1.3	Process vs procedure	20
2.2	Tree Recursion	20
2.3	Example: Counting change	23
2.4	Orders of Growth	24
2.4.1	Pros/cons of order of growth	25
2.5	Exponentiation	26
2.6	Greatest common divisor	27
2.6.1	Lame's Theorem	28

2.7	Testing for primality	28
2.7.1	Searching for divisors	28
2.7.2	Fermat test	29
2.7.3	Probabilistic methods	30

1 The Elements of Programming

Programming language serves multiple purposes.

- Means for instructing a computer to perform tasks
- Serves as a framework to organise our ideas about **processes**

We need to look at how the language combines single ideas to form more complex ones - which is done via the following:

- **Primitive expressions** - simplest entities the language is concerned with
- **Means of combination** - compound elements are built from simpler ones
- **Means of abstraction** - compound elements can be named and manipulated as units

Two elements of programming:

- **Data** - "stuff" to manipulate
- **Procedures** - descriptions of rules for manipulating data
 - Also described as methods
 - Used for combining and abstracting procedures and data

1.1 Expressions

In Lisp, entering an **expression** into an interpreter will cause the interpreter to *evaluate the expression*.

Combining expressions representing numbers with expressions representing primitive procedures form compound expressions

- Represents the application of the procedure to those numbers

(+ 21 26 12 7 35)

The above is an example of an expression representing the application of procedures to those numbers.

- Formed by delimiting a list of expressions within parentheses in order
- **Operator** - leftmost element in the list
- **Operands** - other elements in the list
- **Value of combination** - apply procedure specified by the operator to the arguments
- **Arguments** - value of the operands

Prefix notation - placing operator to the left of the operands.

- Advantages:
- Accommodates procedures with arbitrary number of arguments

```
(+ 21 35 12 7)
>>> 75
(* 25 4 12)
>>> 1200
```

- No ambiguity since operator is always the leftmost element and the entire combination is delimited by the parentheses
- Allows combinations to be nested

```
(+ (* 3 5) (- 10 6))
>>> 19
```

- * No limit to depth of nesting and to the overall complexity of the expression the Lisp interpreter can evaluate

Pretty printing - each long combination is written so that the operands are aligned vertically which display the structure of the expression.

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))

(+ (* 3
    (+ (* 2 4)
        (+ 3 5)))
    (+ (- 10 7)
        6))
```

Order of interpretation - regardless of complexity of expression.

- Read expression from terminal
- Evaluate expression
- Print results
- Aka *read-eval-print loop*

1.2 Naming and the Environment

Using a name to refer to computational objects

- Name identifies a *variable* whose *value* is the object

```
(define size 2)
```

That has associated the name *size* to the value 2. We can use this name to refer to the value 2 from now on.

```
size  
>>> 2  
(* 5 size)  
>>> 10
```

define is the simplest means of abstraction.

- Allows the use of simple names to refer to the results of compound operations

Due to the complex structure of computational objects, it becomes difficult to remember their details and this is where using names help.

- Through the use of names, we build larger programs from smaller procedures.

Environment - memory that stores the *name-value* pairs.

- The one we use now is known as the **global environment**.

1.3 Evaluating Combinations

The interpreter takes the following steps to evaluate a combination:

- Evaluate the subexpressions of the combination
- Apply the procedure that is the value of the operator to the arguments

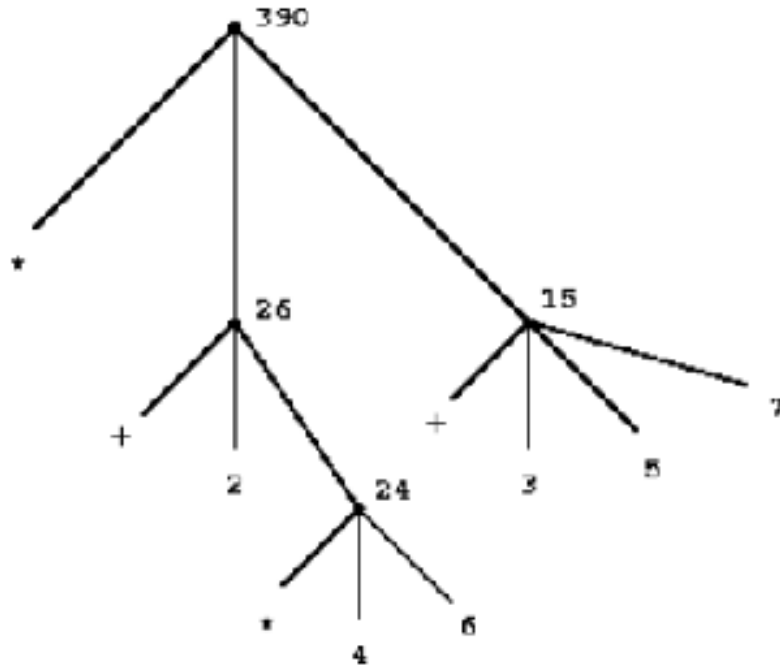
Evaluation is *recursive* in nature, meaning that in order for the operation to occur, it must invoke itself.

- In this scenario, for evaluation to occur, it must first evaluate all expressions

```
(* (+ 2 (* 4 6))  
  (+ 3 5 7))
```

In the example above, the combination can be represented by a tree.

- Each combination is represented by a node with branches corresponding to the operator and the operands of the combination stemming from it
- **Terminal nodes** represent either operators or numbers
- Values of operands precolate upward
 - Moving from terminal nodes and combining at higher and higher levels
 - **Tree accumulation** - process of accumulating



Due to the recursive nature of evaluation, we end up evaluating expressions, not combinations.

- Values of numerals are the numbers that they name.
- Values of built-in operators are the machine instruction sequences that carry out the corresponding operations.
- Value of other names are the objects associated with those names in the environment.

For the above rules of evaluation, the second rule is a special case of the third rule - the symbols $+$ / $*$ are stored in the global environment and are associated with the sequence of machine instructions as their *values*.

The evaluation rule does not handle definitions - instead of applying `define` to two arguments such as `(define x 3)`, `define` associates `x` to the value of 3.

- Definitions are **not** combinations
- Exception to the general evaluation rule

Special forms - exceptions to the general evaluation rule.

- Each special form has its own evaluation rule

1.4 Compound Procedures

Procedure definitions - compound operations with names and referred to as a unit

To illustrate the idea of procedure definition, we think of the procedure as an instruction:

To square something, multiply it by itself

Then, we express that in our language as such:

```
(define (square x) (* x x))
```

We have created a *compound procedure* with the name *square*.

- The procedure represents the operation of multiplying something by itself
- Thing to multiplied has the local name *x*

The general form of procedure definition is:

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

- *<name>* - symbol to be associated with the procedure definition in the environment
- *<formal parameters>* - names used within the body of the procedure to refer to the corresponding arguments of the procedure
- *<body>* - expression that yield the value of the procedure application when the formal parameters are replaced by the actual arguments to which the procedure is applied
- *<name>* and *<formal parameters>* are grouped within parantheses
- As they would be in an actual call to the procedure being defined.

With *square* defined, we can now use it:

```
(square 21)
>>> 441
(square (+ 2 5))
>>> 49
```

We can even use it as a building block in defining other procedures.

```
(define (sum-of-squares x y)
  (+ (square x) (square y)))

(sum-of-squares 3 4)
>>> 25
```

1.5 The Substitution Model for Procedure Application

The interpreter applies the same process as primitive procedures to procedure application.

The body of the procedure is evaluated with each formal parameter is replaced by the corresponding argument.

```
(f 5)
```

And say that **f** has the following definition, it gives new meaning to the above procedure call.

```
(define (f x) (sum-of-squares (+ x 1) (* x 2)))

(f 5)
(sum-of-squares (+ 5 1) (* 5 2))
```

As such, the problem is now the evaluations of a combination with two operands and an operator, **sum-of-squares**.

With the new expanded form, we evaluate the parameters to 6 and 10 respectively.

Then, after replacing **f** with its body definition of **sum-of-squares**, we will continue to substitute each procedure with its body - in this case now, we will substitute **sum-of-squares** with its body comprising of **square**.

```
(sum-of-squares 6 10)
(+ (square 6) (square 10))
```

Then, we apply the body of **square** to obtain our final step.


```
(+ (square 6) (square 10))  
(+ (* 6 6) (* 10 10))
```

And now that we are left with only primitive operations, we will finally reduce it.

```
(+ 36 100)  
>>> 136
```

This process applied is known as the *substitution model* for procedure application.

- Way of thinking of procedure application, not an overview of how interpreters work
- More than 1 evaluation model

1.5.1 Applicative order vs normal order

Evaluating all operators and operands and then applying the procedure to the arguments is not the only method of evaluation.

An alternative is to only evaluate operands until their values are needed.

- Substitute operand expressions for parameters until it obtained an expression involving only primitive operators and then perform evaluation

```
(f 5)  
(sum-of-squares (+ 5 1) (* 5 2))  
(+ (square (+ 5 1) (* 5 2)))  
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))  
(+ (* 6 6) (* 10 10))  
(+ 36 100)  
>>> 136
```

The thing to note with this evaluation model is that some procedures might be evaluated twice, like `(+ 5 1)` and `(* 5 2)`.

Normal-order evaluation - "fully expand and then reduce"

- Contrast to **applicative-order evaluation** - "evaluate the arguments and then apply"

Lisp uses applicative-order evaluation.

- Due to additional efficiency obtained from avoiding repeated evaluations of the same expressions
- Normal-order evaluation becomes much more complicated to deal with after leaving the realm of procedures that can be modelled by substitution

1.6 Conditional Expressions and Predicates

Case analysis - construct where we make tests and perform different operations depending on the result of said test.

$$|x| = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -x & \text{if } x < 0 \end{cases}$$

For instance, the above declares the function of *absolute*. In order to replicate this in Lisp, we use a special form known as `cond`.

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

`cond` general form:

```
(cond (<p1> <e1>)
      (<p2> <e2>)
      ...
      (<pn> <en>))
```

- **Clauses** - parenthesized pairs of expressions `<p> <e>`

- **Predicate** - $\langle p \rangle$ - expression whose value is interpreted as **true** or **false**
- **Consequent expression** - $\langle e \rangle$ - value to be given if the matching predicate is **true**
- Evaluated in order of clauses, if **p1** is false, then it moves on to **p2** and so forth
- If none of the predicates are true, the value of **cond** is undefined

Alternative for writing absolute-value procedure:

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

Expressed in English as

+begin_{quote} If **x** is less than 0 return -**x**; otherwise return **x** #+end_{quote}
else - used in place of a predicate in the final clause of a **cond**

- **cond** returns its value if all other clauses have been bypassed (all other predicates are false)

Another alternative way of writing absolute-value procedure:

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

if - restricted type of conditional used when there're precisely two cases in the case analysis.

- General form:

```
(if <predicate> <consequent> <alternative>)
```

- Evaluation - starts with $\langle predicate \rangle$, if **true**, return $\langle consequent \rangle$, else, return $\langle alternative \rangle$

1.6.1 Logical composition operators

Alongside `<`, `=`, `>`, there are other logical composition operators.

1. `and`

- Evaluates left-to-right order
- If any `<e>` evaluates to **false**, entire expression is **false**
- If all `<e>` evaluate to **true**, only then will expression be **true**
- Special form, not procedure

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

```
(and (> x 5) (< x 10))
```

The above expression represents a condition that a number `x` must be in the range `5 < x < 10`.

2. `or`

- Evaluates left-to-right order
- If any `<e>` evaluates to **true**, the whole expression is **true**
- If all `<e>` evaluates to **false**, the whole expression is **false**
- Special form, not procedure

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

```
(define (>= x y)
  (or (> x y) (= x y)))
```

3. `not`

- If `<e>` evaluates to **false**, expression is **true** and vice versa

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

1.7 Example: Square Roots by Newton's Method

Conceptually, procedures are akin to mathematical functions. However, what sets procedures apart from mathematical functions is the fact that they have to be effective.

Mathematically, we can represent the square-root function as

$$\sqrt{x} = \text{the } y \text{ such that } y \geq 0 \text{ and } y^2 = x \quad (1)$$

However, the definition, while accurate mathematically, does not define a procedure computationally.

```
(define (sqrt x)
  (the y (and (>= y 0)
              (= (square y) x))))
```

Herein lies the contrast between (mathematical) functions and procedures is a reflection of the distinction between describing properties of things and describing how to do things.

- Also referred to as the distinction between *declarative knowledge* and *imperative knowledge*.
- In mathematics, we are concerned with declarative descriptions (what is)
- In computer science, we are concerned with imperative descriptions (how to)

Leading to the use and definition of **Newton's method of successive approximations**.

1.7.1 Newton's method of successive approximation

We start with a guess y for the value of the square root of a number x .

To obtain a better guess (closer to the actual square root), we use the following manipulations:

$$\text{Average } y \text{ with } \frac{x}{y} \quad (2)$$

This average then becomes the new guess or y and we continue till we hit a "good enough" criteria which would be the case where the guess is as close to the square root as possible, often within some minute fractional difference.

We represent this definition in lisp as follows:

```

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)
                  x)))

(define (improve guess x)
  (average guess (/ x guess)))

(define (average x y)
  (/ (+ x y) 2))

(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))

(define (sqrt x)
  (sqrt-iter 1.0 x))

```

1.8 Procedures as Black-Box Abstractions

`sqrt-iter` is recursive - procedure is defined in terms of itself.

Any large program can be dissected into parts.

- Each procedure accomplishes an identifiable task that can be used as a module in defining other procedures.
- We regard each sub-procedure as a "black box"
 - We are not concerned with *how* it works, we only care that it computes the result
 - Aka **procedural abstraction**
 - The procedure definition should be able to suppress detail
 - User does not need to have written the procedure but can still use the code like a blackbox.

1.8.1 Local names

Meaning of a procedure should be independent of the parameter names used by its author.

- The parameter name must be local to the body of the procedure

- If the parameters are not local to the bodies of the procedure, the parameter might be confusing to the developer/user of the procedure
- **Formal parameter** - special role in the procedure definition - it doesn't matter what name the formal parameter has.
 - This is known as a **bound variable**
 - Procedure definition *binds* its formal definition
 - Definition remains unchanged if a bound variable is consistently renamed throughout the definition.
 - If a variable is not bound, it is *free*
 - **Scope** - set of expressions for which a binding defines a name

1.8.2 Internal definitions and block structure

Some procedures might involve functions with the same name but have different implementations.

- To remedy this issue, we create a procedure to have internal definitions that are local to that procedure

```
(define (sqrt x)
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.001))
  (define (improve guess x) (average guess (/ x guess)))
  (define (sqrt-iter guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x)))
  (sqrt-iter 1.0 x))
```

- **Block structure** - solution for the name-packaging problem proposed
 - To further add to the solution, since **x** is bound in the definition of **sqrt**, the nested procedures are all in the scope of **x**
 - Don't need to pass **x** to each procedure - **x** is a free variable in the internal definitions
 - Aka *lexical scoping*

2 Procedures and the Processes They Generate

The knowledge we possess now is akin to understanding the rules of chess but having no experience playing the game.

- To build upon this knowledge, we have to be able to /visualize the process generated by various types of procedures.

Procedure - pattern for *local evaluation* of a computational process.

- Specifies how each stage of the process is built upon the previous stage
- Make statements about the *global* behaviour of a process whose local evolution has been specified by a procedure

2.1 Linear Recursion and Iteration

Factorial is defined as the following function:

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdots 3 \cdot 2 \cdot 1 \quad (3)$$

A common pattern for calculating factorial is realising that $n!$ is equal to n times $(n - 1)!$ for any positive integer n .

$$n! = n \cdot [(n - 1) \cdot (n - 2) \cdots 3 \cdot 2 \cdot 1] = n \cdot (n - 1)! \quad (4)$$

We can translate this program to lisp:

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
(factorial 6)
```

An alternate approach to computing factorials would be to describe a rule that computing $n!$ would state that we would first multiply 1 by 2, then take the result of that multiplication and multiply it to 3 and so forth till we reach n

- A more formal definition of the rule would be that we maintain a *running product*, together with a *counter* that counts from 1 to n .

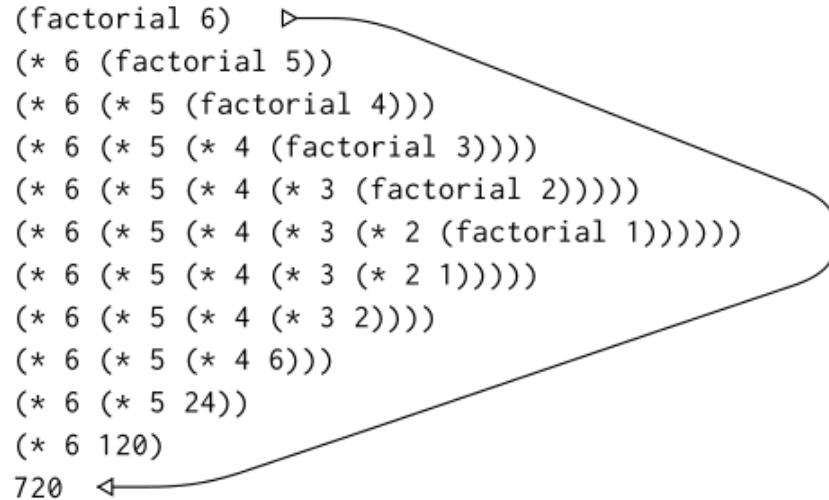


Figure 1: Linear recursive process

- As the calculations progress, the following changes are made to the *running product* and *counter*

$$product \leftarrow counter \times product \quad (5)$$

$$counter \leftarrow counter + 1 \quad (6)$$

- We also stipulate that $n!$ is the value of the product when the counter exceeds n

```

(define (factorial n)
  (fact-iter 1 1 n))
(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product)
                  (+ counter 1)
                  max-count)))

```

Much like the other process defined, we will produce a specific tree for the way the process is executed:

```
(factorial 6)  ▷
(fact-iter 1 1 6)
(fact-iter 1 2 6)
(fact-iter 2 3 6)
(fact-iter 6 4 6)
(fact-iter 24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720  ◁
```

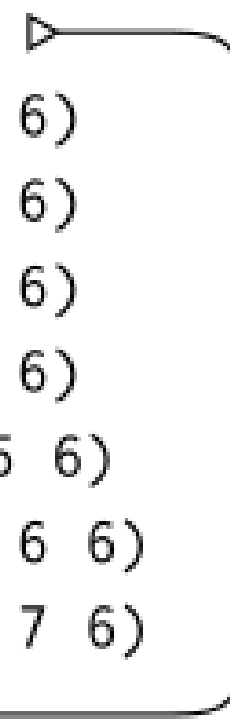


Figure 2: Linear recursive process

While the two processes perform the same operation, they carry different **shapes** and evolve quite differently over time.

2.1.1 Linear recursive process

- For the first process, the substitution model reveals a shape of expansion followed by contraction
 - **Expansion** - occurs as the process builds up a chain of *deferred operations*
 - **Contractions** - occurs as the operations as the operations are performed
 - **Recursive process** - process characterized by a chain of deferred operations
 - Interpreter keeps track of the operations to be performed later on
 - * Length of the chain of deferred multiplications == the amount of information needed to keep track of it, growing linearly with n
 - * Aka **proportional to n**
 - This process is known as a *linear recursive process*
- "Hidden information" is maintained by the interpreter and not contained in program variables.
 - These indicate where the process is
 - The longer the chain of deferment, the more information must be maintained

2.1.2 Linear iterative process

- For the second process, the process does not grow and shrink
- At each step, we keep track of *product*, *counter* and *max-count*
- Aka *iterative process* - any process whose *state* can be summarized by a fixed number of *state variables* along with a *fixed rule* that describes how the state variables should be updated as the process moves from state to state
 - Might include an **optional** end test that specifies conditions in which the process should terminate

– Aka *linear iterative process*

- Program variables provide a complete description of the state of the process at any point
- If the process is stopped mid-way, we would have to resume computation by providing the values of the program variables before the program was terminated.

2.1.3 Process vs procedure

Recursive *process* is different from recursive *procedure*

- **Recursive procedure** - syntactic fact that the procedure definition refers to the procedure itself
 - Aka, it's a procedure that is recursive
- **Recursive process** - how the process evolves, not the syntax
 - For instance, with the second implementation of the **factorial** procedure, it is described as being a *recursive procedure* with a *linear process* due to the shape it takes on
- The distinction is made due to the implementations of other languages
 - These languages are designed to take any interpretation of any recursive procedure to consume an amount of memory that grows with the number of procedure calls - regardless of whether or not the process described is *iterative*
 - These languages can only describe iterative processes using special-purpose "**looping-constructs**"
 - This is not how Racket/Scheme is implemented
 - **Tail-recursion** - process of executing an iterative process that is described by a recursive procedure in constant space
 - * Everything is expressed using ordinary procedure call mechanisms

2.2 Tree Recursion

Tree recursion - another common pattern of computation.

Fibonacci sequence - illustration of tree recursion

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Fibonacci numbers are defined by the following rule:

$$\text{Fib}(n) = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ \text{Fib}(n - 1) + \text{Fib}(n - 2) & \text{otherwise.} \end{cases}$$

Figure 3: Fibonacci numbers rules

This can be translated to lisp as such

```
(define (fib n)
  (cond ((= n 0) 0)
        (= n 1) 1)
  (else (+ (fib (- n 1)) (fib (- n 2))))))
```

The evolved process of this procedure would look like a tree.

- Branches are *split* into two or more at each level

This is an inefficient method of computing Fibonacci numbers

- Performs a lot **redundant** computation
 - For instance, `(fib 3)` is computed multiple times and it's being used frequently, which means it is wasting resources to re-compute the same value
- $\text{Fib}(n)$ grows exponentially with n
 - Growth closest to:

$$\phi/\sqrt{5} \tag{7}$$

Where

$$\phi = \frac{1 + \sqrt{5}}{2} \approx 1.6180 \tag{8}$$

Interestingly enough, ϕ is the *golden ratio* where

$$\phi^2 = \phi + 1 \tag{9}$$

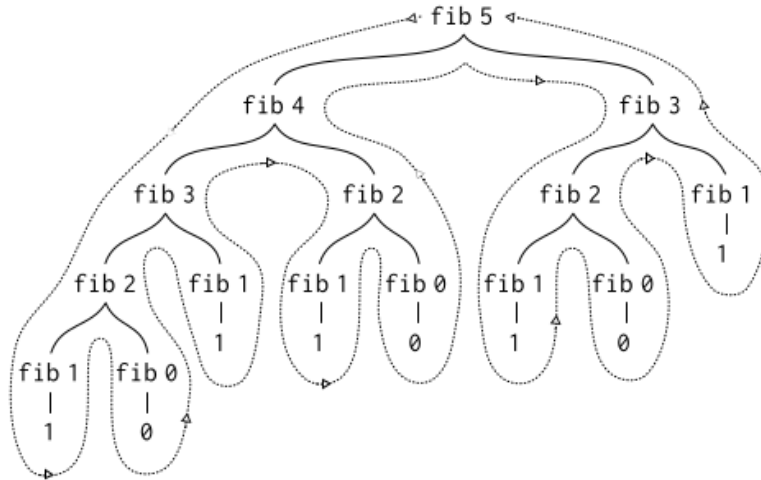


Figure 4: Fibonacci Tree

- *Space* required grows **linearly** with input
 - * Keep track only of which nodes are above us in the tree at any point in the computation

Number of steps required - proportional to number of nodes in the tree

Space required - proportional to maximum depth of the tree

The process demonstrated is a **recursive** process with a **recursive** procedure.

- Fibonacci can be described using an **iterative** process
 - Use a pair of integers, **a** and **b** initialised to $\text{Fib}(1) = 1$ and $\text{Fib}(0) = 0$ and apply the following transformations:

$$a \leftarrow a + b \quad (10)$$

$$b \leftarrow a \quad (11)$$

```
(define (fib n)
  (fib-iter 1 0 n))
(define (fib-iter a b count)
```

```
(if (= count 0)
    b
    (fib-iter (+ a b) a (- count 1))))
```

- This version of the Fibonacci sequence uses linear iteration
 - The difference in number of steps required by each method
 - * One is linear in n
 - * One is growing as fast as $\text{Fib}(n)$

In spite of the drawbacks of tree recursion, tree recursion is not useless.

- When operating on **hierarchically structured** data rather than numbers, tree recursion is a *powerful and natural* tool.
- It also serves to help us understand and design programs, while performance is not top notch, the readability and ease of understanding certainly is
 - To create the iterative process, we needed to recast as an iteration with three state variables

2.3 Example: Counting change

An interesting recursive problem is to find the number of ways to make change.

- Think of the types of coins available as arranged in some order
- The number of ways to change a using n kinds of coins is equal to
 - Number of ways to change amount a using all but the first kind of coin, plus
 - Number of ways to change $a - d$ using all n kinds of coins
 - * d - denomination (face value of a note) of the first kind of coin
- Algorithm is defined as such
 - $/a/ == 0 \rightarrow 1$ way
 - $/a/ < 0 \rightarrow 0$ way
 - $/n/ == 0 \rightarrow 0$ way

```

(define (count-change amount) (cc amount 5))
(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc amount
                      (- kinds-of-coins 1))
                  (cc (- amount
                        (first-denomination
                          kinds-of-coins))
                      kinds-of-coins))))))
(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))

```

- **Note:** `denomination` assumes the order of largest to smallest but any order is fine

`count-change` generates a tree-recursive process - similar to `fib`

- There's no iterative alternative (like in `fib`)

"Smart compiler" - proposed to get the best of both worlds - *clarity* of tree-recursion and *efficiency*

2.4 Orders of Growth

Order of growth - describing the difference in rate of consumption of computational resources

- Obtains a gross measure of the resources required by a process as the input gets larger

n - parameter that measures the size of the problem

$R(n)$ - amount of resources required for size n

- n represents more than just the input of the function, eg.
 - *Square root* - n can represent the number of digits accuracy required

- *Matrix multiplication* - n can represent the number of rows in the matrices
- n encapsulates properties of a problem - which can be used to analyze a given process
- For $R(n)$ to perform a **fixed number of operations** at a time, the time required will be proportional to the number of elementary machine operations performed
- $R(n)$ has order of growth of $\Theta(f(n))$ - $R(n) = \Theta(f(n))$
 - For large n , $R(n)$ will be sandwiched between $k_1 f(n)$ and $k_2 f(n)$
 - * k_1 and k_2 are not related to n

Eg.

- Factorial - *Linear recursive process* - number of steps grows *proportionally* to n
 - Steps grows as $\Theta(n)$
 - Space grows as $\Theta(n)$
- Factorial - *Linear iterative process* - number of steps grows *proportionally* to n
 - Steps grows as $\Theta(n)$
 - Space grows as $\Theta(1)$ - **constant**
- Fibonacci - *Tree recursive process*
 - Steps grows as $\Theta(\phi^n)$
 - Spaces grows as $\Theta(n)$

2.4.1 Pros/cons of order of growth

- *Con* - provides crude description of behavior
 - n^2 , $1000n^2$ and $3n^2 + 10n + 17$ all have the *same* order of growth - $\Theta(n^2)$
- *Pro* - provides indication of changes in behavior as *size* of program changes

- For a **linear** process, *doubling* size will roughly *double* the amount of resources used
- For an **exponential** process, each increment in problem size multiplies resource usage by a *constant factor*

2.5 Exponentiation

Calculating the exponential of a given number can be defined as follows:

$$b^n = b \cdot b^{n-1} \quad (12)$$

$$b^0 = 1 \quad (13)$$

This translates to the following procedure:

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

For this approach, we require a linear number of steps and linear amount of space ($\Theta(n)$).

We can formulate an iterative process for this procedure as well.

```
(define (expt b n)
  (expt-iter b n 1))
(define (expt-iter b counter product)
  (if (= counter 0)
      product
      (expt-iter b
                  (- counter 1)
                  (* b product)))))
```

For this approach, we require a linear number of steps ($\Theta(n)$) but a constant amount of space ($\Theta(1)$).

For even faster computations, we look to optimising the calculation of exponents with powers that are even.

$$b^2 = b \cdot b \quad (14)$$

$$b^4 = b^2 \cdot b^2 \quad (15)$$

$$b^8 = b^4 \cdot b^4 \quad (16)$$

Merging this approach with our original solution, we can come up with the following rule for quickly calculating exponents.

$$\text{if } n \text{ is even, } b^n = (b^{\frac{n}{2}})^2 \quad (17)$$

$$\text{if } n \text{ is odd, } b^n = b \cdot b^{n-1} \quad (18)$$

```
(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
(define (even? n)
  (= (remainder n 2) 0))
```

`fast-expt` grows logarithmically with n in both space and number of steps.

- To compute b^{2^n} , one more multiplication is needed than computing b^n
 - Size of exponent doubles with every new multiplication
 - Number of multiplications required for an exponent of n grows about as fast as the logarithm of n to the base of 2

The advantages of a logarithmic order of growth to a linear order of growth is apparent as the size of the input increases.

- If n is 1000, `fast-expt` requires 14 multiplications while `expt` requires 1000 multiplications

2.6 Greatest common divisor

Greatest common divisor of two integers a and b is defined to be the largest integer that divides both a and b without a remainder.

Knowing how to compute the GCD will aid in reducing rational numbers to the lowest terms.

- To reduce a rational number to the lowest terms, divide the numerator and denominator by the GCD

One approach is to factor them and find the common factors.

Or use an algorithm developed (Euclid's Algorithm):

- If r is the remainder when a is divided by b , the common divisors of a and b are precisely the same as the common divisors of b and r .

- $\text{GCD}(a, b) = \text{GCD}(b, r)$

- Can be used to reduce the problem of computing a GCD to the problem of computing the GCD of smaller and smaller pairs of integers

- Reduce the pair until the remainder is 0

$$\text{GCD}(206, 40) = \text{GCD}(40, 6) = \text{GCD}(6, 4) = \text{GCD}(4, 2) = \text{GCD}(2, 0) = 2$$

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

- Euclid's Algorithm has logarithmic growth

2.6.1 Lamé's Theorem

If Euclid's Algorithm requires k steps to compute the GCD of some pair, then the smaller number in the pair must be greater than or equal to the k^{th} Fibonacci number

- Get an order-of-growth estimate for Euclid's Algorithm
 - Let n be the smaller of the two inputs to the procedure
 - If the process takes k steps, $n \geq \text{Fib}(k) \approx \varphi^k / \sqrt{5}$
 - # of steps k grows as the logarithm (to the base φ) of n
 - Order of growth is $\Theta(\log n)$

2.7 Testing for primality

2.7.1 Searching for divisors

- To test if a number is prime, find the number's divisors
 - A number is prime if and only if n is its own smallest divisor

- Start with a divisor of 2
- If n is not prime, it must have a divisor $\leq \sqrt{n}$
 - # of steps to find n as prime has an order of growth of $\Theta(\sqrt{n})$

```
(define (smallest-divisor n) (find-divisor n 2))
(define (divide? a b) (= (remainder b a) 0))
(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else find-divisor n (+ test-divisor 1))))
(define (prime? n)
  (= n (smallest-divisor n)))
```

2.7.2 Fermat test

- $\Theta(\log n)$ primality test

Fermat's Little Theorem: If n is a prime number and a is any positive integer less than n , then a raised to the n^{th} power is congruent to $a \% n$

- **Congruent modulo:** if both numbers have the same remainder when divided by n
- **Modulo:** remainder of a number a when divided by n
-

To apply the Fermat test to find primality, the following steps can be taken:

Given a number n , pick a random number $a < n$ and compute the remainder of a^n modulo n .

If the result is not equal to a , then n is certainly not prime.

If it is a , repeat the test with a different value.

The more tests that are performed through this method, the higher certainty we can assert that n is a prime.

```

;; The exp is the only value that changes in expmod
;; If the exp == 0, return 1 to start the calculations
;; If the exp is even, square the result of the previous expmod calculation and divide
;; If the exp is odd, multiply the base (random number) to the result of the previous c
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)
         (remainder (square (expmod base (/ exp 2) m))
                     m))
        (else
         (remainder (* base (expmod base (- exp 1) m))
                     m))))

;; For the test, we will generate a random number that is less than the target number
(define (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1)))))

;; To calculate if the value is a prime number, we will attempt the fermat test several
(define (fast-prime? n times)
  (cond ((= time 0) true)
        ((fermat-test n) (fast-prime? n (- times 1)))
        (else false)))

```

2.7.3 Probabilistic methods

Algorithms that exist where the answers are not guaranteed to be correct but with sufficient iterations, the chance of error of this algorithm is arbitrarily small so that it is negligible.

- The Fermat's test is an example of a probabilistic algorithm
 - For every n that fails the test, we are certain that n is not prime
 - But for every n that passes the test, we are not certain that n is prime
 - There are modifications that can be made to the Fermat's test that can help to improve this assertion but there's no way to guarantee that n is a prime

- * There are some instances of numbers that pass the Fermat's test but are not prime but these are rare cases
- If n passes the test for 2 randomly generated numbers, then the chances of n being prime is better than $\frac{3}{4}$