Games like chess or monopoly have a small set of rules. Knowledge of the rules does not necessarily make one skilled at the game.

- We lack the knowledge of which moves are worth making (i.e. which procedures are worth defining).
- We lack the experience to predict the consequences of making a move (i.e. executing a procedure).

The ability to visualize the consequences of the actions under consideration is crucial to becoming an expert programmer.

A procedure is a pattern for the local evolution of a computational process.

It specifies how each stage of the process is built upon the previous stage.

A procedure is a pattern for the local evolution of a computational process.

It specifies how each stage of the process is built upon the previous stage.

Ideally, one would like to be able to make statements about the overall or global, behavior of a process whose local evolution has been specified by a procedure.

This is very difficult to do in general, just like it is difficult to understand the end-game in chess.

A procedure is a pattern for the local evolution of a computational process.

It specifies how each stage of the process is built upon the previous stage.

Ideally, one would like to be able to make statements about the overall or global, behavior of a process whose local evolution has been specified by a procedure.

This is very difficult to do in general, just like it is difficult to understand the end-game in chess.

One way to understand the processes is to measure the computational resources that a procedure uses.

```
> (define (plus_v1 a b)
    (
     if (= a 0)
        b
        (+ 1 (plus_v1 (- a 1) b))
    )
  )
```

```
> (define (plus_v2 a b)
    (if (= a 0)
         b
         (plus_v2 (- a 1) (+ b 1))
     )
   )
```

Both these procedures are would correctly return the sum of two natural numbers.

However, their behaviour is significantly different. The computational resources they use are also significantly different.

Let's look at the process generated by these two procedures in order to compute the sum of 3 and 6.

$$
\begin{aligned}
&(\mathit{plus}\ 3\ 6)\\
\Longrightarrow\ &(+\ 1\ (\mathit{plus}\ 2\ 6))\\
\Longrightarrow\ &(+\ 1\ (+\ 1\ (\mathit{plus}\ 1\ 6)))\\
\Longrightarrow\ &(+\ 1\ (+\ 1\ (+\ 1\ (\mathit{plus}\ 0\ 6))))\\
\Longrightarrow\ &(+\ 1\ (+\ 1\ (+\ 1\ 6)))\\
\Longrightarrow\ &(+\ 1\ (+\ 1\ 7))\\
\Longrightarrow\ &(+\ 1\ 8)\\
\Longrightarrow\ &9
\end{aligned}
$$

The expansion occurs as the process builds up a chain of deferred operations.
The contraction occurs as the operations are actually performed.

This type of process, characterized by a chain of deferred operations, is called a recursive process.
If the length of the chain of deferred multiplications grows linearly with the input, then such a recursive process is called a linear recursive process.
Here the space needed by the procedure is $O(a)$ and time needed is also $O(a)$, where a is the first operand.

$$
\begin{aligned}
&(plus\ 3\ 6)\\
&\Longrightarrow (plus\ 2\ 7)\\
&\Longrightarrow (plus\ 1\ 8)\\
&\Longrightarrow (plus\ 0\ 9)\\
&\Longrightarrow 9
\end{aligned}
$$

This variant of plus does not shrink or grow in size. The amount of space used remains fixed and does not depend on the size of the input. This is an example of an iterative process.

This variant also happens to be a linear iterative process, as the number of steps grow linearly w.r.t the input.

An iterative process is one whose state can be summarized by

- a fixed number of state variables,
- a fixed rule that describes how the state variables should be updated as the process moves from state to state and
- an (optional) end test that specifies conditions under which the process should terminate.

- In the iterative case, the program variables provide a complete description of the state of the process at any point.

  If we stopped the computation between steps, all we would need to do to resume the computation is to supply the interpreter with the values of the three program variables.

- In the iterative case, the program variables provide a complete description of the state of the process at any point.

  If we stopped the computation between steps, all we would need to do to resume the computation is to supply the interpreter with the values of the three program variables.

- In the recursive case, there is some additional "hidden" information, maintained by the interpreter and not contained in the program variables.

  This hidden information indicates "where the process is" in negotiating the chain of deferred operations.
  The longer the chain, the more information must be maintained.

When we describe a procedure as recursive, we are referring to the syntactic fact that the procedure definition refers (either directly or indirectly) to the procedure itself.

When we describe a process as say linearly recursive, we are speaking about how the process evolves, not about the syntax of how a procedure is written.

As a procedure, the second variant of plus is recursive but as a process it is iterative.

In several languages, the looping constructs like while, for, do, repeat are specially constructed so as to describe the iterative processes.

Scheme will execute an iterative process in constant space, even if the iterative process is described by a recursive procedure.

In several languages, the looping constructs like while, for, do, repeat are specially constructed so as to describe the iterative processes.

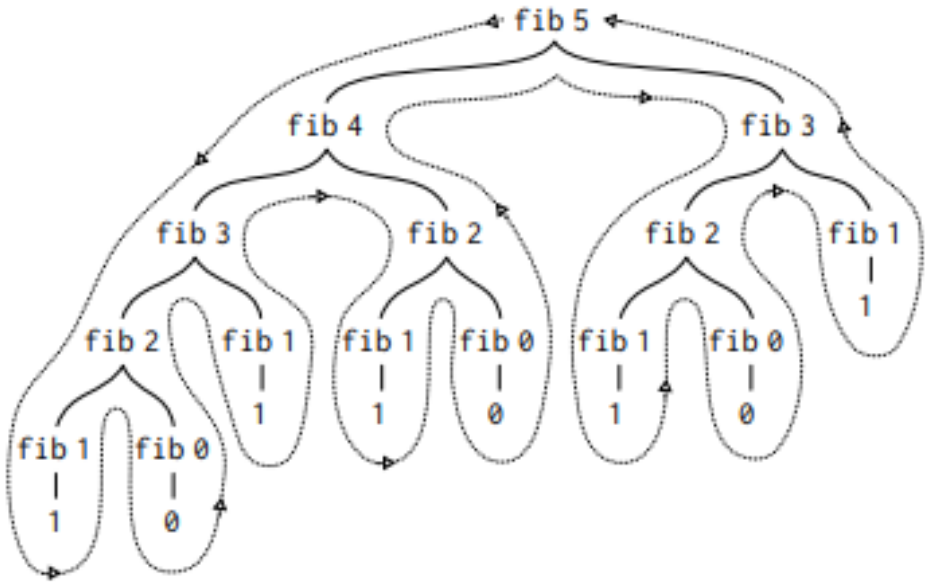
Scheme will execute an iterative process in constant space, even if the iterative process is described by a recursive procedure.

An implementation with this property is called tail-recursive. With a tail-recursive implementation, iteration can be expressed using the ordinary procedure call mechanism, so that special iteration constructs are useful only as syntactic sugar.

One implementation of Fibonacci numbers showcases tree-recursive process.

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

The process associated with computation of (fib 5)



Alternatively, one can also compute Fibonacci numbers in a bottom up fashion which would showcase linear iterative process.

```
(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))))
```

To formulate the iterative algorithm requires noticing that the computation could be recast as an iteration with three state variables.