

# Agenda

- Langage d'expressions préfixées
- Les fonctions
- **Programmer avec des images / Animations**
- Programmer par récurrence
- Les listes (chainées)
- Les calculs itératifs
- Type abstraits et généralisation
- Les arbres binaires

- Racket propose des **teachpacks**, bibliothèques pour l'enseignement.
- Ce chapitre est une introduction au teachpack **2htdp/image**.
- Ce teachpack permet de construire des **images** composées de formes élémentaires, pour produire une **scène** statique.
- Nous pourrons plus tard programmer la **simulation** graphique **animée** d'un Monde virtuel. Une horloge sera automatiquement mise en place pour scander l'évolution du Monde, par exemple tous les 1/28 sec... Une animation n'est en effet pas autre chose qu'une suite d'images.
- Le teachpack **image** est déjà intégré dans le teachpack **valrose** !

# Images géométriques de base

- *Demandez la doc avec l'aide en ligne de DrRacket...*

```
> (rectangle 40 20 'solid "blue")
```



```
> (rectangle 40 20 'outline "blue")
```



```
> (circle 20 'solid "red")
```



```
> (ellipse 50 20 'outline "red")
```



```
> (line 50 10 "blue")
```



```
> (text "Hello !" 32 "red")
```

**Hello !**

```
> (star 40 'solid "green")
```



- Les Couleurs

- Une couleur peut se représenter par une **chaîne de caractères** :



- ou par une structure représentant un **mélange RGB** avec la fonction (make-color r g b) qui attend des arguments dans [0,255] :

- > (rectangle 50 20 'solid (make-color 0 0 255)) 

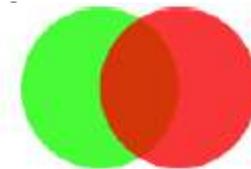
- La couleur **jaune** s'obtiendrait comme mélange de rouge et de vert :

- (rectangle 50 20 'solid (make-color 255 255 0)) 

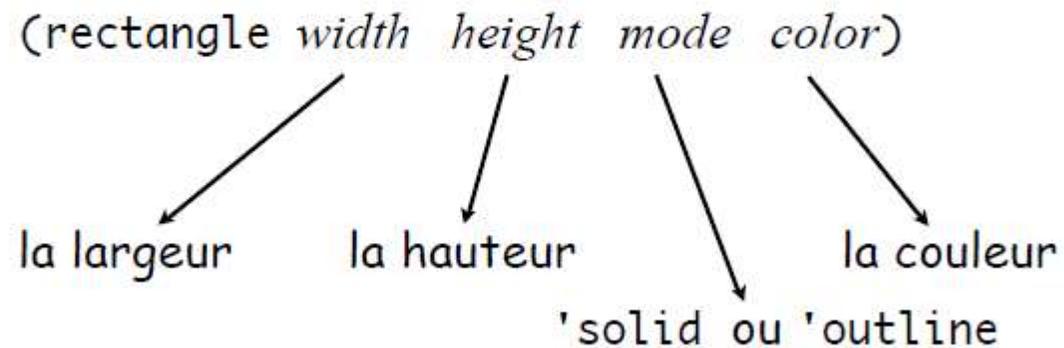
- Un quatrième argument optionnel représente la **transparence**. Il doit être choisi entre 0 (transparent) et 255 (opaque).

- (underlay/xy (circle 50 'solid (make-color 0 255 0)) 5

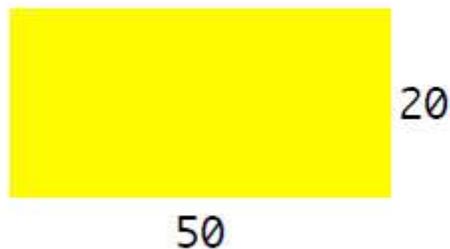
- (circle 50 'solid (make-color 255 0 0 200)))



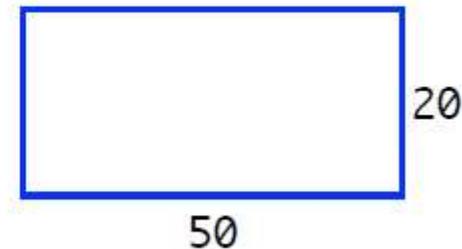
**Le Rectangle** qui permet de construire le fond d'une scène !



*Creates a width by height rectangle, filled in according to mode and painted in color color*



(rectangle 50 20 'solid "yellow")



(rectangle 50 20 'outline "blue")

- A quoi sert l'accent aigu [on le prononce "quote"] ?

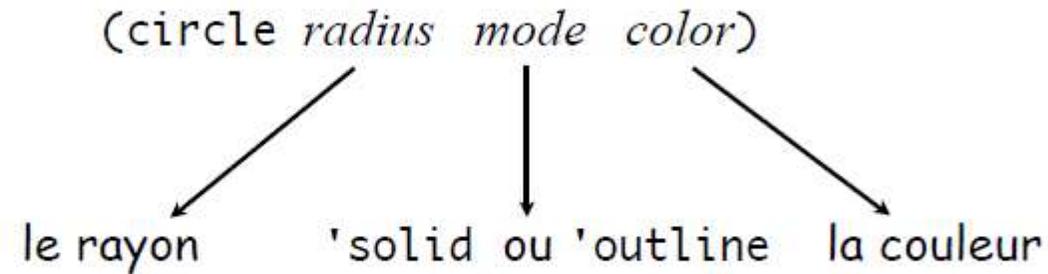
➔ A exprimer que l'expression qui suit ne doit pas être évaluée !

```
Welcome to DrRacket, version 5.3
> pi
#i3.141592653589793
> bonjour
ERROR : undefined identifier : bonjour
> 'bonjour
bonjour
> (+ 1 2)
3
> '(+ 1 2)
(+ 1 2)
```

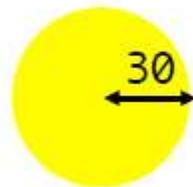
*Il croit que bonjour est une variable et il cherche sa valeur !*

*J'exprime que ce n'est pas une demande de calcul mais une donnée à l'état brut.*

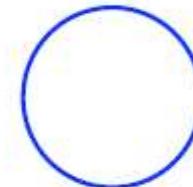
## Le Cercle



*Creates a circle or disk of radius radius, filled in according to mode and painted in color color*

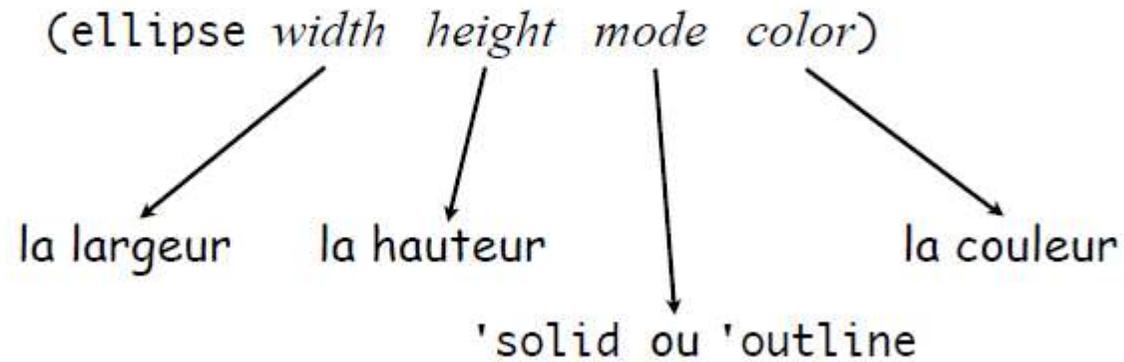


`(circle 30 'solid "yellow")`

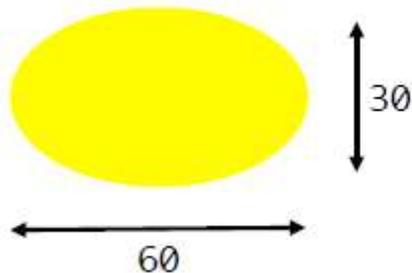


`(circle 30 'outline "blue")`

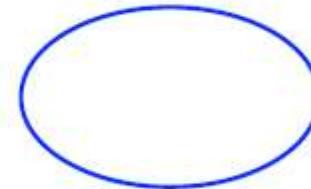
## L'Ellipse



*Creates a width by height ellipse, filled in according to mode and painted in color color*

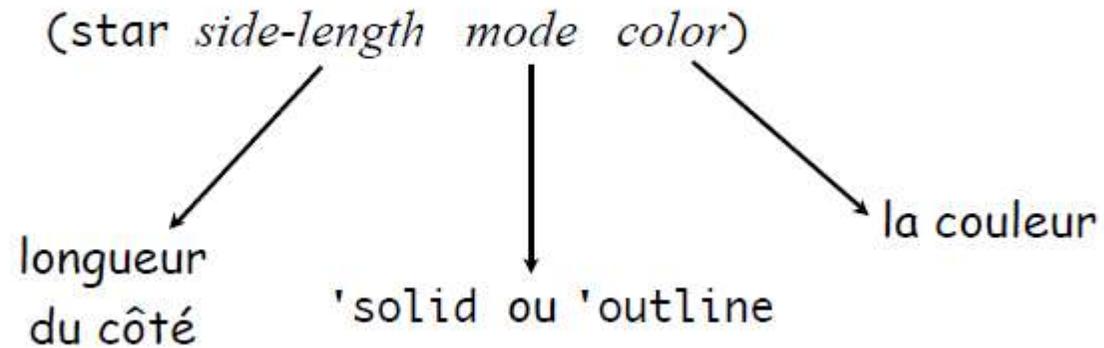


`(ellipse 60 30 'solid "yellow")`

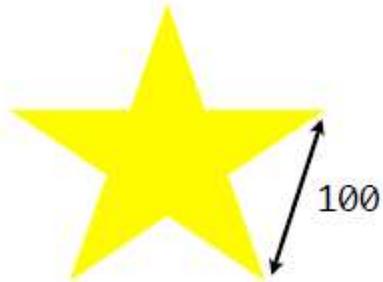


`(ellipse 60 30 'outline "blue")`

## L'Etoile



*Creates a multi-pointed star with 5 points, the side-length argument determines the side length of the enclosing pentagon.*

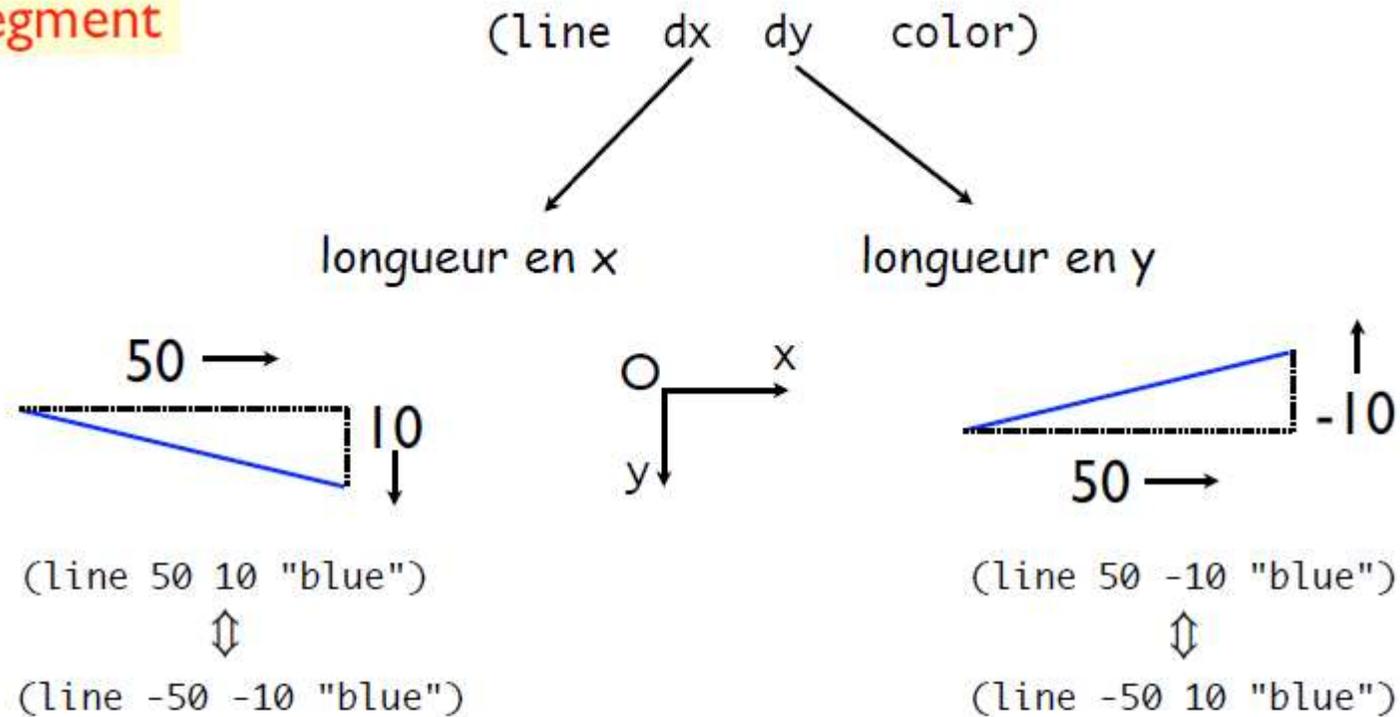


`(star 100 'solid "yellow")`



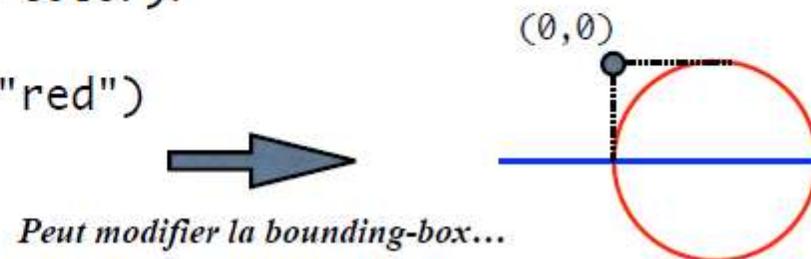
`(star 100 'outline "blue")`

## Le segment

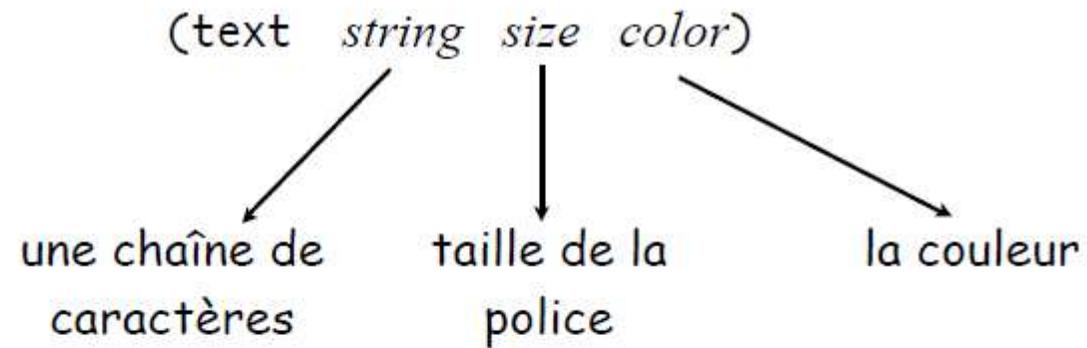


- Pour ajouter à une image `img` un segment joignant  $A(x_1, y_1)$  à  $B(x_2, y_2)$ , utilisez `(add-line img x1 y1 x2 y2 color)`.

```
(add-line (circle 40 'outline "red")  
-50 40 80 40  
"red")
```



## Le Texte



*Creates an image of the text string at point size size and painted in color color.*

# Hello !

(text "Hello !" 96 "blue")

- La police de caractère n'est pas réglable. Seule la **taille** et la **couleur** le sont. Ou alors utilisez **text/font**...
- Le premier argument [le texte] doit être de type *string*. Il peut être intéressant de **transformer un nombre en une chaîne de caractères** :

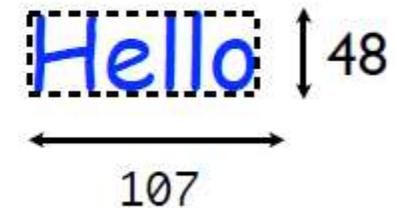
```
> (number->string 1789)
"1789"
> (number->string 1789 2) ; en binaire [base 2]
"11011111101"
> (number->string 1789 16) ; en hexadécimal [base 16]
"6fd"
```

- La fonction (**format** str x1 x2 ...) s'utilise de la même manière que (**printf** str x1 x2 ...) mais elle retourne une chaîne de caractères au lieu d'afficher !

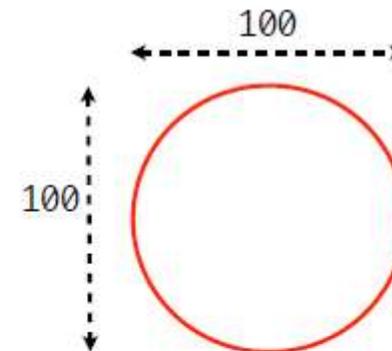
```
> (text (format "pi = ~a" pi) 48 "blue")
pi = 3.141592653589793
```

- On peut obtenir les **dimensions d'une image** :

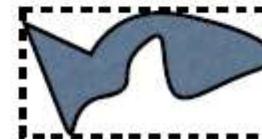
```
> (define T (text "Hello" 48 "blue"))  
> T  
Hello  
> (image-width T)  
107  
> (image-height T)  
48
```



```
> (define C (circle 50 'outline "red"))  
> (image-width C)  
100  
> (image-height C)  
100
```



- Une image est contenue dans un rectangle transparent : sa **bounding box**.



## Superposition centrée d'images [underlay]

- La fonction `(underlay img1 img2 ...)` permet de combiner plusieurs images en une seule [img1 au-dessous de img2, etc] en les alignant par leurs centres.

```
(underlay (rectangle 160 40 'solid "gray")  
          (circle 10 'solid "black"))
```

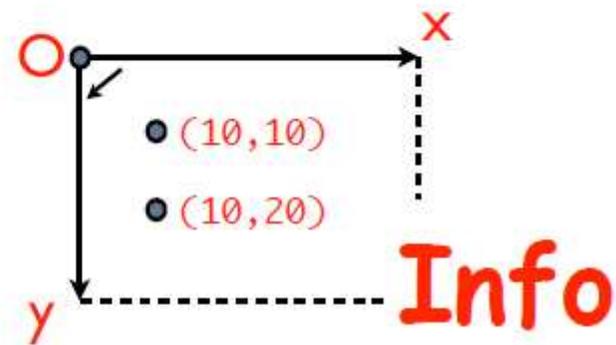
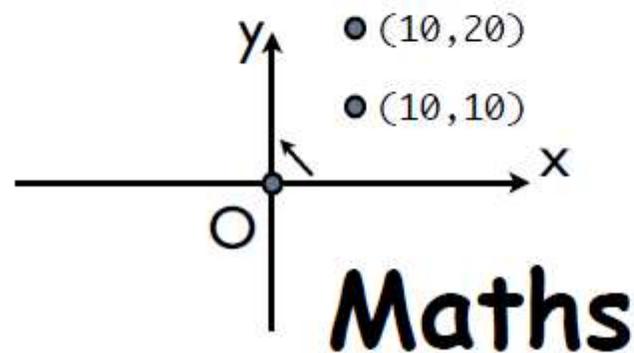


- Comment construiriez-vous celles-ci ?...



## Superposition relative [underlay/xy]

- ATTENTION : la plupart des langages de programmation graphiques n'utilisent pas les axes mathématiques usuels !



- La fonction `(underlay/xy img1 x y img2)` superpose `img2` au-dessus de `img1`, mais en décalant `img2` de `x` pixels vers la droite et de `y` pixels vers le bas par-rapport au coin haut gauche de `img1`.

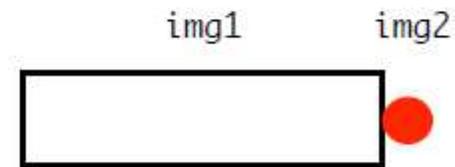
```
(underlay/xy (rectangle 160 40 'solid "gray")  
80 20  
(rectangle 160 40 'solid "black"))
```



## Alignement horizontal centré [beside]

- La fonction `(beside img1 img2 ...)` permet de combiner plusieurs images en une seule [img1 à gauche de img2, etc] en les alignant par leurs centres.

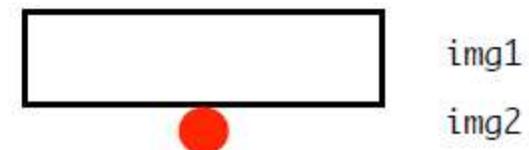
```
(beside (rectangle 160 40 'outline "black")  
        (circle 10 'solid "red"))
```



## Alignement vertical centré [above]

- La fonction `(above img1 img2 ...)` permet de combiner plusieurs images en une seule [img1 en haut de img2, etc] en les alignant par leurs centres.

```
(above (rectangle 160 40 'outline "black")  
        (circle 10 'solid "red"))
```

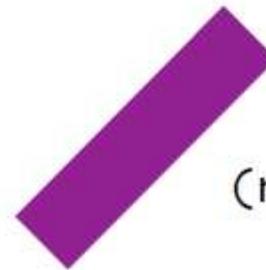


## Rotation d'une image [rotate]

- Il est possible de faire tourner une image d'un angle  $\alpha$  exprimé en degrés avec la fonction `(rotate  $\alpha$  img)`. La rotation aura lieu dans le sens inverse des aiguilles d'une montre.



r



(rotate 45 r)

## Changement d'échelle d'une image [scale]



r



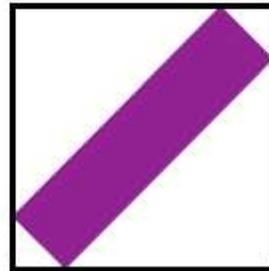
(beside r (scale 2 r))

## Encadrement d'une image [frame]

- Chaque image est contenue dans un rectangle invisible qui l'entoure : sa bounding box. La fonction `(frame img)` permet de le visualiser !



r



`(frame (rotate 45 r))`

## Extraction d'une sous-image [crop]

- La fonction `(crop x y larg haut img)` permet d'extraire la portion d'image de `img`, dont le coin haut gauche est à la position `(x,y)`, et dont les dimensions sont `larg` et `haut`. C'est le **cropping** !



`c = (circle 40 'solid "red")`



`(crop 40 0 40 40 c)`

## Le placement avec cropping [place-image]

- La fonction (`place-image img1 x y img2`) superpose `img1` au-dessus de `img2`, mais en plaçant le centre de `img1` au point `(x ; y)` dans le repère haut gauche de `img2`. En général, `img2` est un rectangle. **CROPPING** !

```
(place-image (circle 100 'solid "black")  
             60 60  
             (rectangle 400 200 'solid "red"))
```



- Ne pas confondre avec (`underlay/xy img1 x y img2`) qui :
  - place le **coin haut gauche de `img1`** par-rapport au coin haut gauche de `img2` alors que `place-image` s'occupe du **centre** de `img1`.
  - ne fais **aucun cropping** et agrandit la *bounding box*.

- ***Lorsque vous travaillez dans un cadre rectangulaire, utilisez toujours `place-image` et évitez `underlay/xy` qui agrandit la *bounding box* !***
- ***La variante `scene+line` de `add-line` fait un cropping automatique !***

# Utilisez de véritables images

- Il est souvent intéressant d'ajouter des images Scheme à un fond en provenance d'une véritable image (photo, Web, etc). Le format PNG est conseillé, mais GIF et JPG sont aussi admis...

```
> (define ballon (bitmap "ballon.png"))  
> (above ballon (beside ballon ballon))
```



# Simuler le mouvement

- Le teachpack **2htdp/universe** de Racket (déjà intégré au teachpack **valrose**) va nous permettre de programmer facilement de petites scènes animées. Applications à la géométrie, à la physique, aux jeux, etc.
- Une **horloge** sera automatiquement mise en place pour scander le temps, et donc l'évolution de la scène, par défaut tous les 1/28 de seconde (donc 28 images par seconde).
- Les animations sont des techniques très utilisées dans les pages Web [par exemple avec HTML5 et son *Canvas*, ou bien *Java* et ses *applets*].
- Une **animation** se construit comme un dessin animé : ce n'est en effet pas autre chose qu'une suite d'images qui défilent très vite pour donner l'illusion du mouvement !

# Simuler le mouvement

- Le programmeur de l'animation devra :
  1. Préciser le **modèle mathématique** de son Monde virtuel. Quel est l'ensemble minimum de **variables** qui le décrivent ? Ceci est CAPITAL.
  2. Préciser comme ce modèle **évolue** à chaque top d'horloge.
  3. Préciser comment ce modèle sera transformé en une **scène** rectangulaire contenant des images.
  4. Préciser (optionnellement) comment ce Monde va **interagir** avec l'utilisateur, via le clavier ou la souris.

# place-image

- La fonction (place-image img x y scene) place le centre de l'image img au point (x,y) par-rapport au coin haut gauche de la scène rectangulaire.

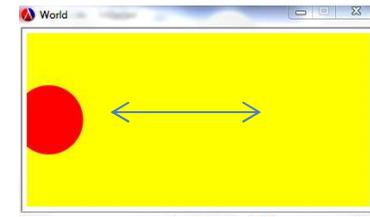
```
(place-image (circle 10 'solid "red") 60 30  
             (rectangle 200 60 'solid "yellow"))
```



- Pour animer le mouvement d'un objet sur un fond fixe, on **placera** l'image de cet objet à un certain point (x,y) sur le fond fixe, les coordonnées (x,y) changeant un tout petit peu à chaque image...
- **N.B.** Pour ajouter une ligne à une scène avec cropping, utilisez la fonction (scene+line img x1 y1 x2 y2 color) plutôt que add-line !

# MOUVEMENT RECTILIGNE SINUSOIDAL - Version 1

- Une balle rouge rebondit sur les parois, avec une vitesse nulle au rebond.
- Il ne s'agit pas d'étudier un problème de *collision* entre un disque et une droite. Ceci serait plus délicat...
- Nous modéliserons plutôt la position du centre de la balle sous la forme d'une **fonction périodique d'un seul paramètre réel  $m$** .
- **La seule variable du modèle mathématique sera donc  $m$ .**

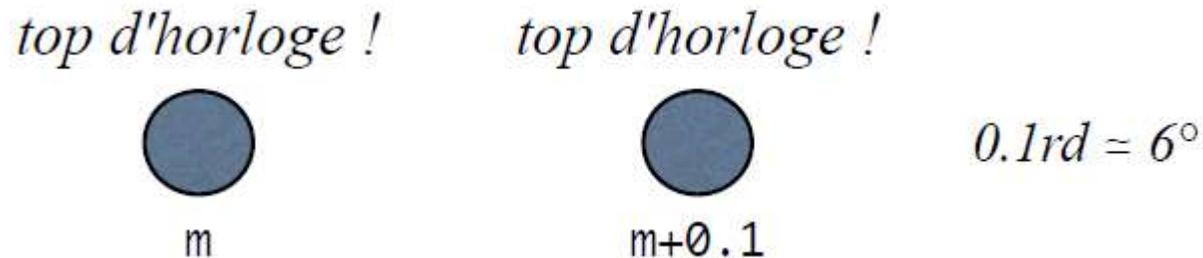


$$(x, y) = (50 + 50 \sin m, 25) \quad m \in [0, +\infty[$$

L'abscisse  $x$  varie entre 0 et 100, tandis que  $y$  est constant.

***Le MONDE n'est autre que l'ensemble des variables indépendantes gouvernant le phénomène !***

- **Mise à jour du Monde.** Etant donné un état  $m$  du modèle mathématique du Monde, quel est l'état suivant [au prochain top d'horloge] ? J'essaye empiriquement :



- On dit qu'on a **discrétisé** la fonction  $m \rightarrow 50+50 \sin(m)$ . On ne la calcule pas en continu, mais par pas discret de 0.1 :

```
(define (suivant m)                      ; Monde → Monde
  (+ m 0.1))
```

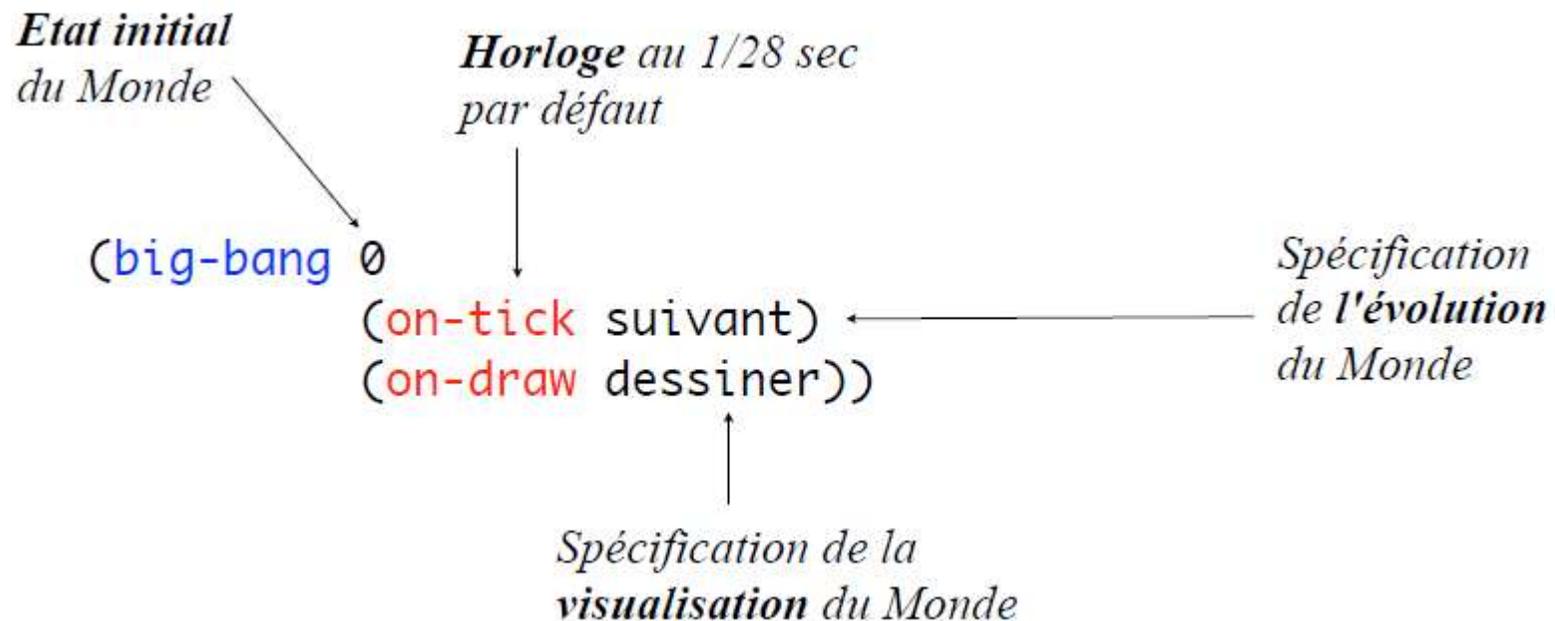
*N.B. On pourrait prendre pour  $m$  le temps  $t$ , mais rester indépendant procure plus de souplesse pour régler la vitesse de la simulation. Pensez que le Monde est décrit par un seul paramètre réel, que ce soit le temps ou autre chose !*

- **Affichage du Monde.** Etant donné un état  $m$  du Monde, comment le transformer en une scène à visualiser ?

```
(define (dessiner m) ; Monde → Scène
  (place-image (circle 10 'solid "red")
    (* 50 (+ 1 (sin m)))
    25
    (rectangle 100 50 'solid "yellow")))
```

- bien voir que l'on ne décrit pas l'animation du Monde par une boucle,
  - on exprime seulement comment le Monde se visualise de manière statique à un moment donné à partir de son modèle mathématique !
- Il reste à voir comment va s'organiser la simulation dans le temps... En réalité, comme dans un dessin animé : image par image...

- **Les tops d'horloge.** Une horloge *invisible* et *silencieuse* va automatiquement être mise en place, à une certaine fréquence, 1/28 sec par défaut. Donc 28 fois par seconde, le modèle mathématique sera mis à jour, et transformé en une scène. Ces deux actions sont bien distinctes et décrites par les fonctions *suivant* et *dessiner* :



---

```

(define (suivant m)                ; Monde → Monde
  (+ m 0.1))

(define (dessiner m)              ; Monde → Scène
  (place-image (circle 10 'solid "red")
    (* 50 (+ 1 (sin m)))
    25
    (rectangle 100 50 'solid "yellow")))

(big-bang 0
  (on-tick suivant)
  (on-draw dessiner))

```

- Tâchez de bien saisir la logique de ce programme. En somme, tout se résume à programmer les fonctions :

<pre> suivant  : <i>Monde</i> → <i>Monde</i> dessiner : <i>Monde</i> → <i>Scène</i> </pre>
--

# MOUVEMENT RECTILIGNE SINUSOIDAL - Version 2

- Un bon programmeur va livrer l'animation dans une seule fonction, avec des variables et fonctions locales.

(define (balle-sinus) ; l'animation est livrée dans une fonction

```
(local [(define LARG 400)
        (define HAUT 200)
        (define SCENE (rectangle LARG HAUT 'solid "yellow"))
        (define RAYON 40)
        (define BALLE (circle RAYON 'solid "red"))
        (define X0 (/ LARG 2))
        (define Y0 (/ HAUT 2))
        (define INIT 0)
        (define dM 0.1)
        (define (suivant m) (+ m dM))
        (define (dessiner m)
          (place-image BALLE (* X0 (+ 1 (sin m))) Y0 SCENE))
        (define (final? m)
          (>= m (* 6 pi))))]
```

```
(big-bang INIT ; Création du monde
  (on-tick suivant) ; chaque 1/28 sec ; les 3 contrôleurs
  (on-draw dessiner LARG HAUT)
  (stop-when final?)))]
```

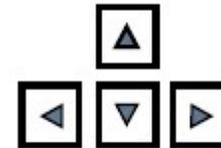
## La gestion des touches du clavier

- Il est parfois intéressant (*jeu vidéo*) d'**interagir avec l'utilisateur**, qui va piloter le jeu en pressant des touches du clavier pendant l'animation.



- Jusqu'à présent, nous avons installé dans une animation :
  - le chef d'orchestre : (big-bang ...)
  - un contrôleur du temps qui fait évoluer le Monde : (on-tick ...)
  - un contrôleur d'affichage qui construit une image : (on-draw ...)
  - un contrôleur de la fin des temps : (stop-when ...)
- Il reste à installer un **contrôleur de clavier** : (on-key ...)
- S'il n'est pas installé, l'animation n'écouterà pas le clavier !

- La fonction  $f$  passée au contrôleur (on-key  $f$ ) va traiter chacun de
- ces evenements clavier :
  - $f : \text{Monde} \times \text{Key} \rightarrow \text{Monde}$
  - ou Key est une chaine de caractères [string] qui peut être
  - une touche usuelle comme "a" "A" "!" " «
  - une touche directionnelle "up", "down", "left", "right". "release" [touche relâchée]



- Les touches se comparent avec ( $\text{key}=? \text{k1 k2}$ ). On pourra par exemple tester la pression sur la barre espace par ( $\text{key}=? \text{k} \text{ " "}$ ).
- Il faut donc programmer une fonction de gestion du clavier (**clavier m key**) qui va calculer un nouveau monde obtenu a partir du Monde  $m$  sur pression de la touche  $key$ . Elle envisagera tous les cas possibles pour  $key$ .

# MOUVEMENT RECTILIGNE SINUSOIDAL - Version 3

- Ajoutons un *reset*, qui remet le Monde à son origine INIT sur pression de la touche r ou R (majuscule ou minuscule) :

```
(define (balle-rect-sin4)
  (local [(define LARG 100) ; hauteur de la scène
          .....
          (define (final? m) ; Monde → Boolean
            (>= m (* 2 pi)))
          (define (clavier m key) ; Monde × Key → Monde
            (if (or (key=? key "r") (key=? key "R"))
                INIT
                m))]
    (big-bang INIT
              (on-tick suivant)
              (on-draw dessiner)
              (stop-when final?)
              (on-key clavier))))
```

# La gestion de la souris

- La communication d'un programme avec l'utilisateur se fait principalement à travers le clavier et la souris. Quid de la souris ?
- La pression d'une touche du clavier gère un *événement-clavier*.
- Quels sont les *événements-souris* ? Ouvrons la doc :

*MouseEvent*

(one-of/c "button-down" "button-up" "drag" "move")

- La fonction *f* passée au contrôleur (on-mouse *f*) va traiter chacun de ces événements-souris !
- ***f* : Monde x integer x integer x mouse-event → Monde**

# MOUVEMENT RECTILIGNE SINUSOIDAL - Version 4

- Réagissons à un **clic sur un bouton de la souris**, qui aura le même effet que le reset obtenu par pression de la touche r ou R du clavier :

```
(define (balle-rect-sin5)
  (local [(define LARG 100)
          .....
          (define (clavier m key)
            (if (key=? key "r") INIT m))
          (define (souris m x y evt)
            (if (mouse=? evt "button-down")
                INIT
                m))]
    (big-bang INIT
              (on-tick suivant)
              (on-draw dessiner LARG HAUT)
              (stop-when final?)
              (on-key clavier)
              (on-mouse souris))))
```

clavier  
*Monde x Key → Monde*

souris  
*Monde x int x int x Mouse-Event → Monde*

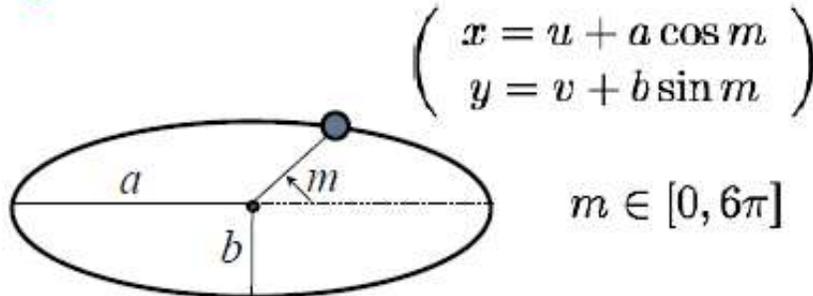
*; je n'écoute que cet évènement !*

# Courbes planes paramétriques

- Vous étudierez en maths les **courbes planes** définies par des **équations paramétriques** : un point mobile  $(x,y)$  dont les coordonnées sont fonctions d'un paramètre réel.
- Dans le cas general :  $x = f(m)$  et  $y = g(m)$  ou  $m \in I \subset \mathbb{R}$ .

$$\frac{(x-u)^2}{a^2} + \frac{(y-v)^2}{b^2} = 1$$

Ellipse



Parabole

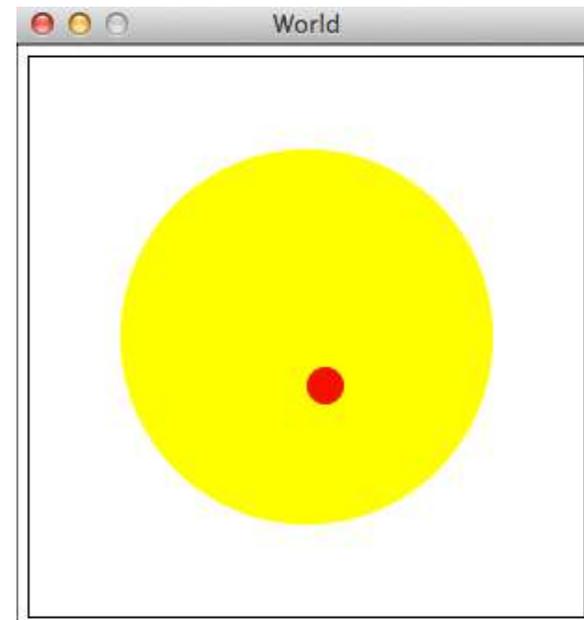
$$\begin{pmatrix} x = m \\ y = m^2 \end{pmatrix}$$

$$m \in [-10, 10]$$



## Animations à plusieurs paramètres

- Jusqu'à présent, notre animation était à 1 paramètre réel. Or la plupart des animations dépendent de plusieurs paramètres. Par exemple de la position  $(x,y)$  d'une balle et du nombre de secondes écoulées, etc.
- Lorsque l'état du phénomène à animer dépend de  $N$  variables indépendantes, on parlera d'une **animation à  $N$  paramètres**.
- **Exemple** : une balle qui essaye de s'échapper d'un disque entouré d'une clôture électrifiée ! L'état du système est déterminé par la seule position  $(x,y)$  de la balle à un instant donné. C'est donc une animation à 2 paramètres.



- (define (electric-ball SIZE R) ; une animation à 2 paramètres !
- (local [(define S (/ SIZE 2))
- (define FOND (underlay (empty-scene SIZE SIZE)
- (circle R 'solid "yellow")
- (circle R 'outline "red"))))
- (define BALLE (circle 10 'solid "red"))
- ; Le Monde m est le point <x;y> à la position de la balle
- (define INIT (make-posn S S)) ; le centre du disque
- (define (distance x1 y1 x2 y2)
- (sqrt (+ (sqr (- x1 x2)) (sqr (- y1 y2)))))
- (define (suivant m) ; Monde --> Monde
- (local [(define x (posn-x m)) (define y (posn-y m))])
- (make-posn (+ x (- (random 9) 4)) ; petit déplacement...
- (+ y (- (random 9) 4)))) ; ...aléatoire
- (define (dessiner m) ; Monde --> Scène
- (place-image BALLE (posn-x m) (posn-y m) FOND))
- (define (final? m) ; Monde --> Boolean
- (>= (distance (posn-x m) (posn-y m) S S) R))]
- (big-bang INIT
- (on-tick suivant)
- (on-draw dessiner)
- (stop-when final?)
- (name "Electric Ball"))))
- **Prog3.3.rkt**

- Un dernier exemple célèbre : **dessin à main levée avec la souris** ! A tout moment nous devons connaître les coordonnées précédentes  $px$ ,  $py$  de la souris, et l'image  $img$  déjà tracée (en réalité un polygone).

```
(define (dessiner-a-la-souris)
  (local [(define SIZE 400)
          ; Le Monde est le triplet (img,px,py)
          (define-struct monde (img px py))
          (define INIT
            (make-monde (rectangle SIZE SIZE 'solid "white") 0 0))
          (define (dessiner m)
            (monde-img m))
          (define (souris m x y evt)
            (if (mouse=? evt "drag")
                (make-monde (add-line (monde-img m)
                                       (monde-px m) (monde-py m) x y "black") x y)
                (make-monde (monde-img m) x y)))]
    (big-bang INIT
              (on-draw dessiner) ; aucune horloge...
              (on-mouse souris))))
```

