RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

VLSI Design:

SKILL

Prof. Dr. P. Fischer

Lehrstuhl für Schaltungstechnik und Simulation
Technische Informatik der Uni Heidelberg

What is Skill and what can it do ?

- SKILL is the shell / control language of cadence
- It is used for
  - Configuration of the environment
    - Definition of library path
    - …
  - Configuration of tools
    - Definition of ShortCuts
    - Definitions of new commands / menu entries
    - …
- Skill allows, for instance, direct access to objects in a layout / schematic view for
  - Scripted creation of shapes / labels / …
  - Automated creation of cells, layouts, symbols
  - Extraction of pad positions, …
  - Definition of parameterized cells (pcells)

How does SKILL look like?

- SKILL – in its 'natural' form – is very similar to LISP ('LISt Processing')
  - Commands have the form `( cmd arg1 arg2 ... )`
  - Data is mostly stored as *lists*
- Operators are possible as well, i.e.
  - `3 + 5` (equivalent to `(plus 3 5)`)
  - `x = 6` (equivalent to `(setq x 6)`)
- A 'C-like' form is possible as well: `cmd( args..)`
  - Note that the `(` must *DIRECTLY* follow `cmd`, i.e. with ***NO*** blank!
- SKILL is caseSENsitTive!
- Comments are started by `;` or enclosed in `/*...*/` (as C)
- SKILL is – normally – interpreted
  - it can also be *compiled* (→ `*.cxt`) end *encrypted*

Where to find Help & Documentation ?

- At http://en.wikipedia.org/wiki/Cadence_SKILL
- On our Linux machines using a Web browser at /opt/eda/IC618/doc/sk...
  There you find for instance

Path	Purpose
sklangref/sklangrefTOC.html	Structure, Basic Commands
sklanguser/sklanguserTOC.html	Data structures
skdevref/skdevrefTOC.html	Routines
skdfref/skdfrefTOC.html	Data objects
sklayoutref/sklayoutrefTOC.html	Layout specific stuff

- Best save some links in your browser!

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

More Help

- You can also use the general cadence help system (/opt/eda/IC618/bin/cdnshelp or via ‚Help‘ buttons)

- There is quite a lot of help in the internet
- If you look around, most questions are answered at the end in a very patient and competent way by

  Andrew Beckett

  from Cadence.

- Thank you Andrew !!!!!

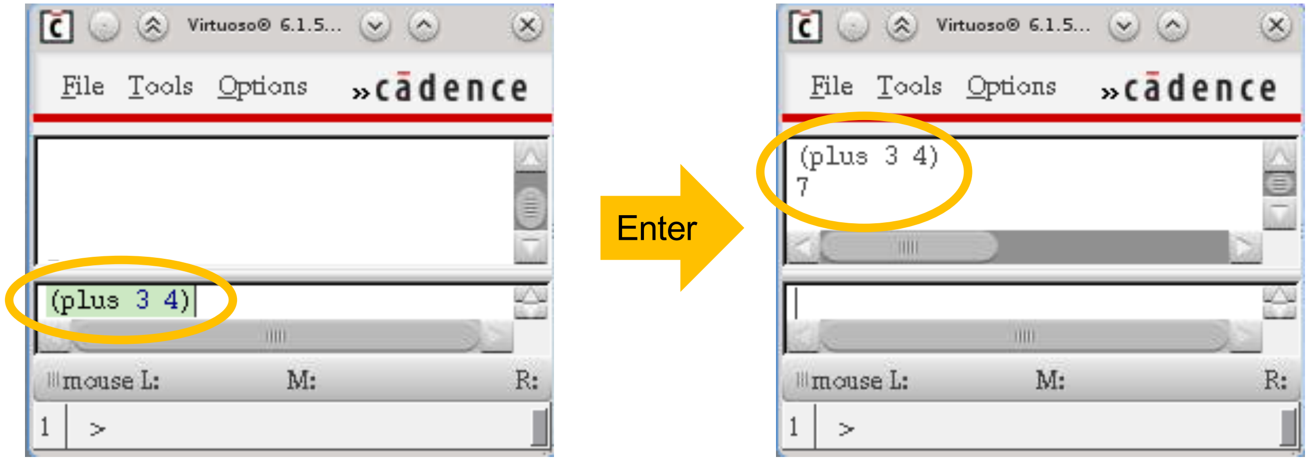
More Help

- You can start an interactive tool from the CIW→Tools→SKILL IDE…
- It contains for instance a linter, step by step processing, breakpoints,…

- You can search for available SKILL functions with the CIW →Tools→SKILL API Finder

How to execute SKILL commands

- You can type commands directly in the Main CIW (Command Interpreter Window):



```
(plus 3 4)
```

```
(plus 3 4)
7
```

- You get back old entries with the **arrow up** key
- You can *select output* with the mouse and *paste it back* to the entry line with the **middle mouse button**

- There seems to be no easy way to clear the CIW

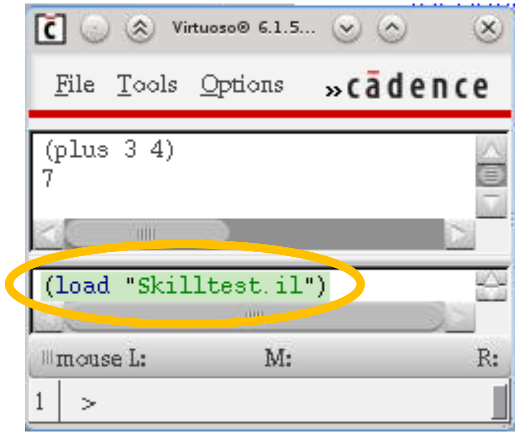
RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Automatic Execution of SKILL

- You can put code in a file (extension *.il) and load the file with **`( load "filename.il" )`**

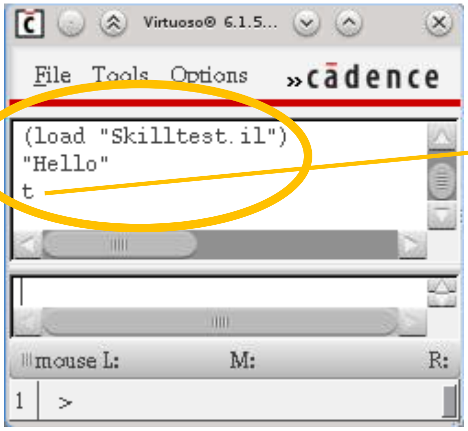
In file "Skilltest.il":

```
(print "Hello")
```



```
(plus 3 4)
7
(load "Skilltest.il")
```

Enter



```
(load "Skilltest.il")
"Hello"
t
```

t: no error

- Code in the file **`.cdsinit`** (in the directory from where you start cadence) is executed at startup of cadence

- In this file, you can
  - Define bindkeys (see exercise 4)
  - Define your own commands
  - Call other skill files

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Basic Objects: Atoms & Lists

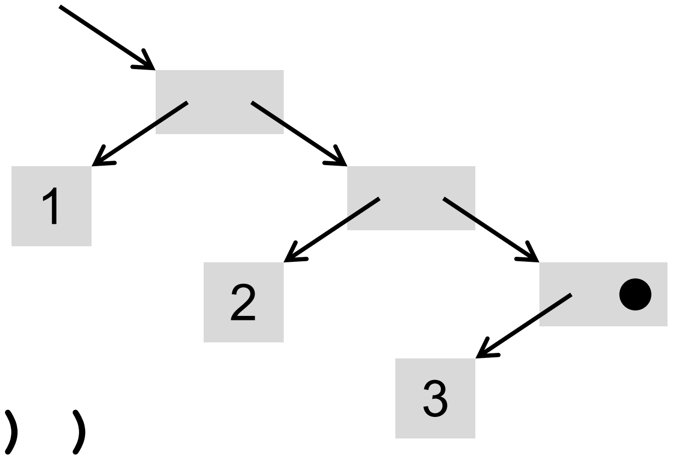
Objects: Atoms and Lists

- An ***atom*** is a simple object:
  - numbers (integers, floats)
  - The boolean values `t` (true) or `nil` (false)
  - pointers (see later)
  - The function `atom` checks if the argument is indeed an atom: `(atom 5)` → `t`
- A ***list*** is a sequence of elements
  - Lists are created by: `( list obj obj ... )` → a list
  - Equivalent: `list( obj obj ... )`
  - Short hand notation: `'( obj obj ...)` (objects are *not* evaluated, works mostly only in top level!)
  - An empty list is `nil` (nil is an atom *and* a list...)

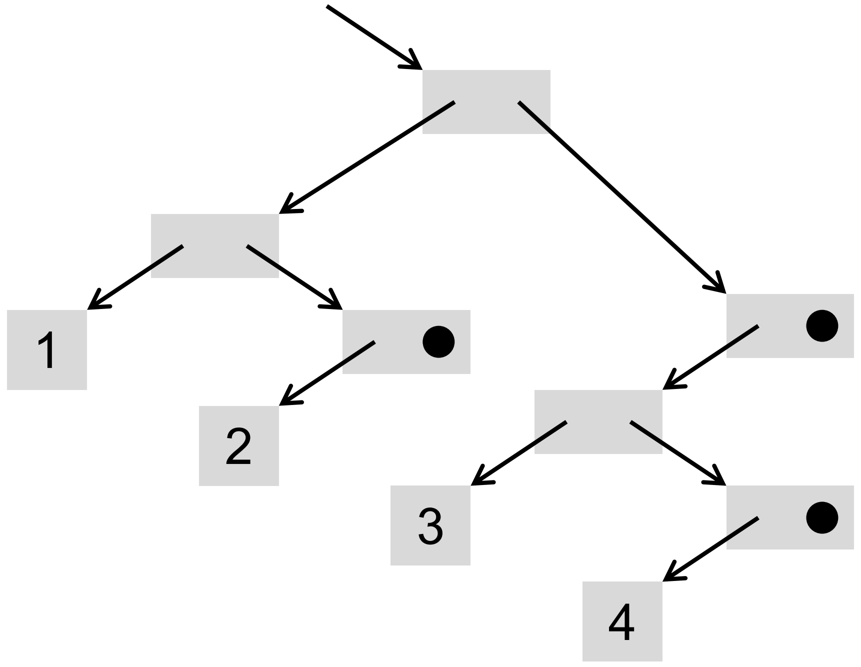
  - Lists are displayed as `(obj obj ...)`
  - Each element can be an atom or another list: `'( (list 1 2) 3)`
  - `(listp obj)` checks if an object is a list

Examples for lists

- `(list 1 2 3)`

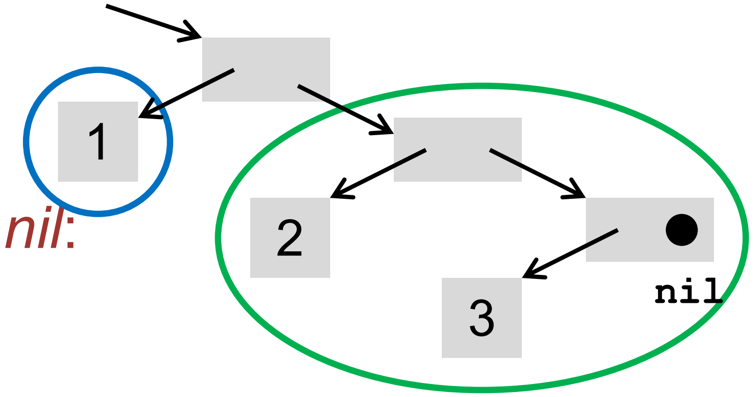


- `(list (list 1 2) (list 3 4) )`



Accessing Parts of lists

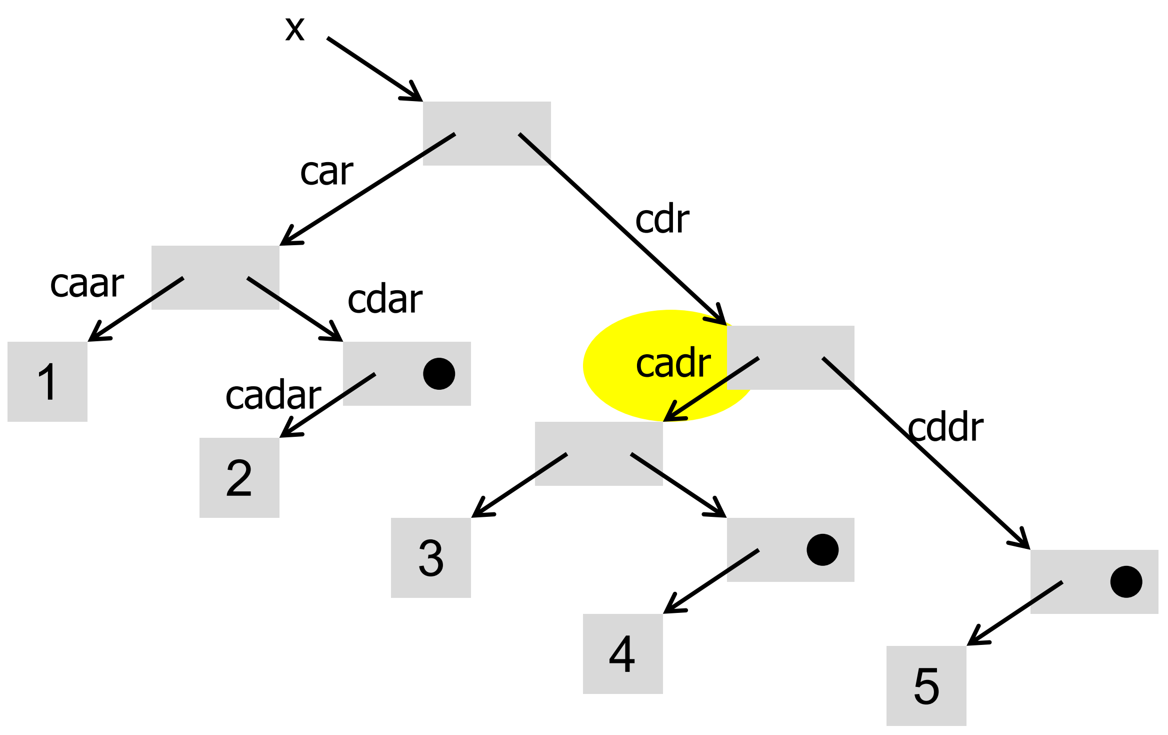
- The first element of a list x is `(car x)`, the rest is `(cdr x)`:
  - `(car '(1 2 3))` → `1`
  - `(cdr '(1 2 3))` → `(2 3)`



- Note: `cdr` always returns a *list* or *nil*:
  - `(car '(1 2))` → `1`
  - `(cdr '(1 2))` → `(2)`
  - `(cdr '(1))` → `nil`
- Extensions for nested lists are `caar`, `cadr`, `cdar`, `cddr`,... (starting evaluation 'at the back'):
  With x = `'( (1 2) (3 4) 5 )`: (see also next page)
  - `(car x)` → `'(1 2)`
  - `(cdr x)` → `'(3 4)`
  - `(caar x)` → `1`
  - `(cdar x)` → `(2)`
  - `(cadar x)` → `2`
  - `(caadr x)` → `3` (note two 'a' !)

Accessing Parts of a List

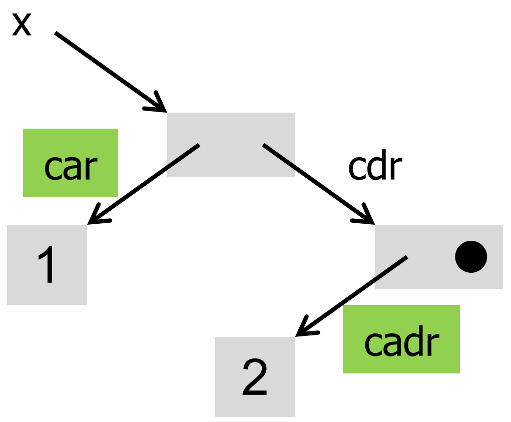
- `(list (list 1 2) (list 3 4) 5 )`



- Note: `(cadr x)` is `(car (cdr x))`

Most important: `car` and `cadr` (not cdr!)

- **`(setq x (list 1 2))`**
- **`(car x)`** and **`(cadr x)`** access the first and second element of a list:



More List Commands

- Get the length of a list (or array / table / ...) (top level!):
  - `(length object)`
  - `(length '(a b c d))` $\rightarrow$ `4`
- Pick the n-th element (first element has index **0**):
  - `(nth index list)`
  - `(nth 2 '(a b c d))` $\rightarrow$ `c`
- Add an element to (the front of) a list:
  - `(cons element list)`
  - `(cons 5 '(a b c d))` $\rightarrow$ `(5 a b c d)`
  - Note: `list` is not changed! To change it, re-assign it:
  - `aa = (cons 5 aa)`
  - You can also **`append`** (two lists!) at the end, but this is slower!

More List Commands

- Check whether an object is in a list:
  - `(member element list)` → `nil` or rest-list
  - `(memq …)`
- Difference: `member` uses ‚`equal`', `memq` uses ‚`eq`'

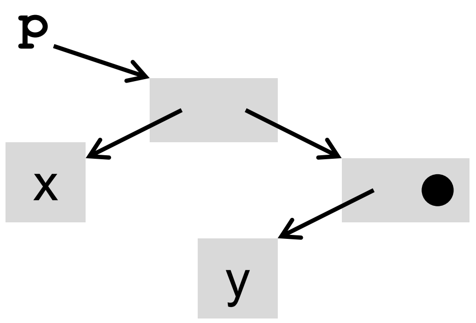
Functions

- Several mathematical functions work on arbitrary number of arguments:
- `(plus 4 5 6)`
- `(times 5 6 7)`
- `(difference 4 3)`
- `(quotient 4 3)` → integer result, if arguments are int!!!, float result if arguments are float
- `(xquotient 4 3)` → integer arguments only!
- `(minus 5)` → -5
- `(float 3)` → 3.0 ; convert integer to float

Points and Rectangles

- A *point* is a *list* of two (float) values
- There is a short hand notation to enter such a list
  - `3.1:4.2` → `(3.1 4.2)`
- To extract the coordinates, one can use
  - `(xCoord p)` equivalent to `(car p)`
  - `(yCoord p)` equivalent to `(cadr p)` (not `cdr` !!!)

  (note capital 'C'!)



- A *rectangle* is a list of two points
  - `list(3:4.2 10:12.1)` → `((3 4.2) (10 12.1))`
- Note: In the database (see later) the first point is always bottom left, i.e. `(xCoord (car p)) < ((xCoord (cadr p))`

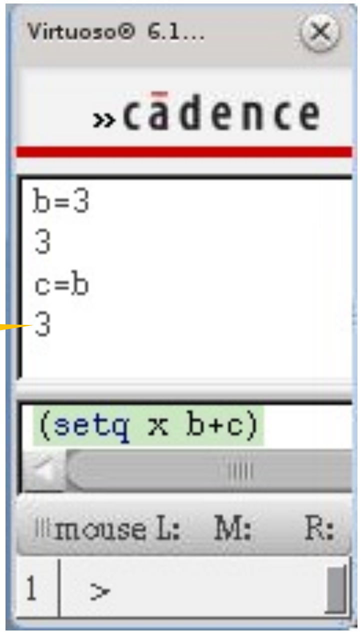
RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Variables

- Variables do *not* need to be *declared*, they are just used
- Assignment can be done with
  - `var = expression`

  or

  - `(setq var expression)`
- Note that expression are *evaluated*:



```
b=3
3
c=b
3
```

```
(setq x b+c)
```

(Difference between `(list …)` and `'(...)`

- `(list …)` evaluates the arguments, `'(..)` does not:

```
(setq x 1.0)
(list x 3.0) → (1.0 3.0)
'(x 3.0) → (x 3.0)
```

- Watch out:

```
list(3:4.2 10:12.1)→ ((3 4.2) (10 12.1))
```

but

```
'(3:4.2 10:12.1)→ ((3:4.2) (10:12.1))
```

- **Better use** `(list …)`! This is safer!

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

CONTROL STRUCTURES

Conditional Execution - if

- ‘Lisp’ version:
  - `( if condition exTrue exFalse)`

- More ‘true’ or ‘false’ statements can be grouped by `{ … }`

- Better readable versions:
  - **(if** condition **then** exT1 exT2 ... **else** exF1 exF2 …**)**
  - **if(** condition **then** exT1 exT2 ... **else** exF1 exF2 …**)** //C-Style

- Examples:
  - `(if t 4 6)` → `4`
  - `(if (greaterp 6 7) 4 6)` → `6`
  - `(if 3+4>3*4 then print(“yes”) else (print “no”))` → `„no“`

Logical Expressions

- Boolean values can be true (**`t`**) or false (**`nil`**)
- Normal operators work: **>**, **<=**, **==**, ..
  - The function equivalents have mostly a 'p' at the end:
  - **`(greaterp 5 4) (leqp 6 7)`**

- Several functions return a Boolean value:

```
• (oddp 7)          →     t
• (plusp -3)        →     nil
• (zerop 0)         →     t
• (floatp 3)        →     nil   ; check data type
```

- WATCH OUT: There are several versions of **`eq, equal`**,… which check content or addresses – see documentation:

```
•p1 = '(1 2)    p2 = '(1 2)
• (equal p1 p2)     →     t     // same values
• (eq    p1 p2)     →     nil   // different objects!
```

More Conditionals

- Execute several expressions if a condition is true:
  - `( when condition ex1 ex2 … )`
- Same if condition is false
  - `( unless condition ex1 ex2 … )`

- Examine several cases:

```
• (cond
     (condition1 expr. expr. …)
     (condition2 expr. expr. …)
     ...
  )
```

Loops

- `(for var initial_value final_value expressions)`
  (loop variable is always incremented by 1!)
- `(while condition expressions)`

- Examples:
  - `(for i 1 9 (print i))` → `123456789`

  -
    ```
    (setq i 1)
    (while i<100 i=2*i (printf "%d " i))
    ```
    → `2 4 8 16 32 64 128`

Very Useful: `foreach`

- All elements of a list can be processed with ‘foreach’:
  `(foreach name list expression)`
  - Variable **`name`** is assigned an element of **`list`** and **`expression`** is executed. This is repeated for all elements of **`list`**.

- Example:
  - **`(foreach x '(1 2 3 5) (println x*x))`**

  $\rightarrow$

```
1
4
9
25
```

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

PROCEDURES

Procedures

- A procedure can be declared with

```
( procedure
   ( name arg1 arg2 …)
   commands
   …
   result of last command is return value
)
```

- Example:
  - `(procedure (square x) (times x x))` // LISP syntax
  - `( square 4 )` → `16`

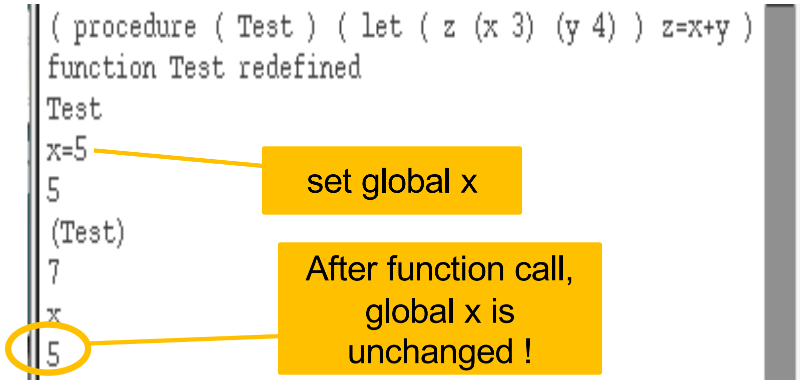
- Alternative syntax:
  - `procedure( square(x) x*x)` // C-like syntax
  - `square(4)` → `16`

(Local Variables)

- When defining procedures, it is *recommended* to declare variables *locally*. This can be done using a **`let`** – block:
- **`( let (`** list of local variables **`)`** commands **`)`**
- The local variables in the list can be
  - Declared by just naming them
  - Initialized using **`(name value)`**
- Example:

```
( procedure
   ( Test ) ;function has no args
   ( let
      ( z (x 3) (y 4) ) ;local variables
      z = x + y
   ) ; end of let
) ; end of procedure
```

- Note: Only one variable with default:
  `(let ((x 1)) … )`



```
( procedure ( Test ) ( let ( z (x 3) (y 4) ) z=x+y )
function Test redefined
Test
x=5
5
(Test)
7
x
5
```

(Functions with Defaults and Named Parameters)

- Function arguments can be assigned a *default value* and can be called *by name* using the `@key` keyword:

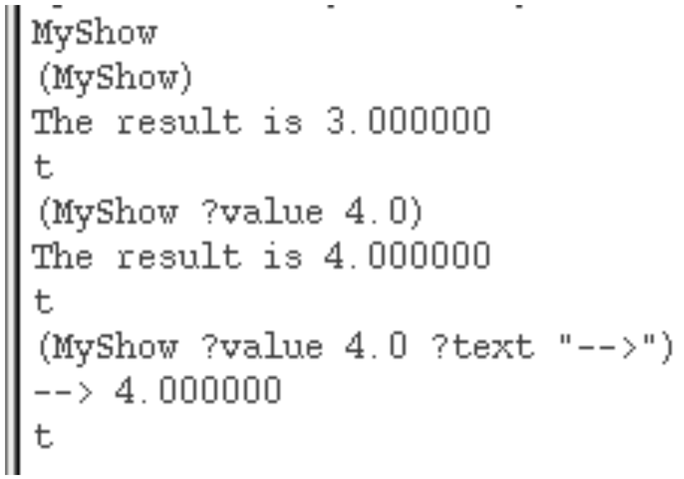
```
( procedure
  ( fname @key (param1 default) … )
  definition …
)
```

- The procedure can then be called with named parameters:

```
( fname ?param1 value ?param2 value…)
```

- Example:

```
( procedure
  ( MyShow @key
    (value 3.0)
    (text "The result is")
  )
  (printf "%s %f\n" text value)
)
```



```
MyShow
(MyShow)
The result is 3.000000
t
(MyShow ?value 4.0)
The result is 4.000000
t
(MyShow ?value 4.0 ?text "-->")
--> 4.000000
t
```

(Formatted & File IO)

- Mostly like in C:

```
(print "Hello")
(println "Hello")
(printf "1+2 is %d\n" (add 1 2))
```

- Format syntax: `%[-][width][.precision] code`
  - - : Left aligned
  - code: d: int, f: float, s: string, c:char, P:point, B: bounding box

- Writing to file:
  - `fp = (outfile "path" [opt])` opt = “a“:append, „w“:write
  - `(fprintf fp "%f\n" 3.141)`
  - `(close fp)` // must close, otherwise empty file !
- Reading from file
  - `fp = (infile "path" [opt])`
  - `x = (fscanf fp "%f %s\n" x c)`

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

THE CADENCE DATABASE

Objects in the DataBase

- All objects used in cell views (wires, pins, labels, shapes, contacts,..) are stored in a ***data base***.
- Access to objects is via their *unique data base object identifier*, or *ID*
- Objects have properties (or 'attributes') / members
- The access operator to the properties is `~>`
- A list of all attributes can be shown with `ID~>?`
- Attributes *& their values* are listed with `ID~>??`

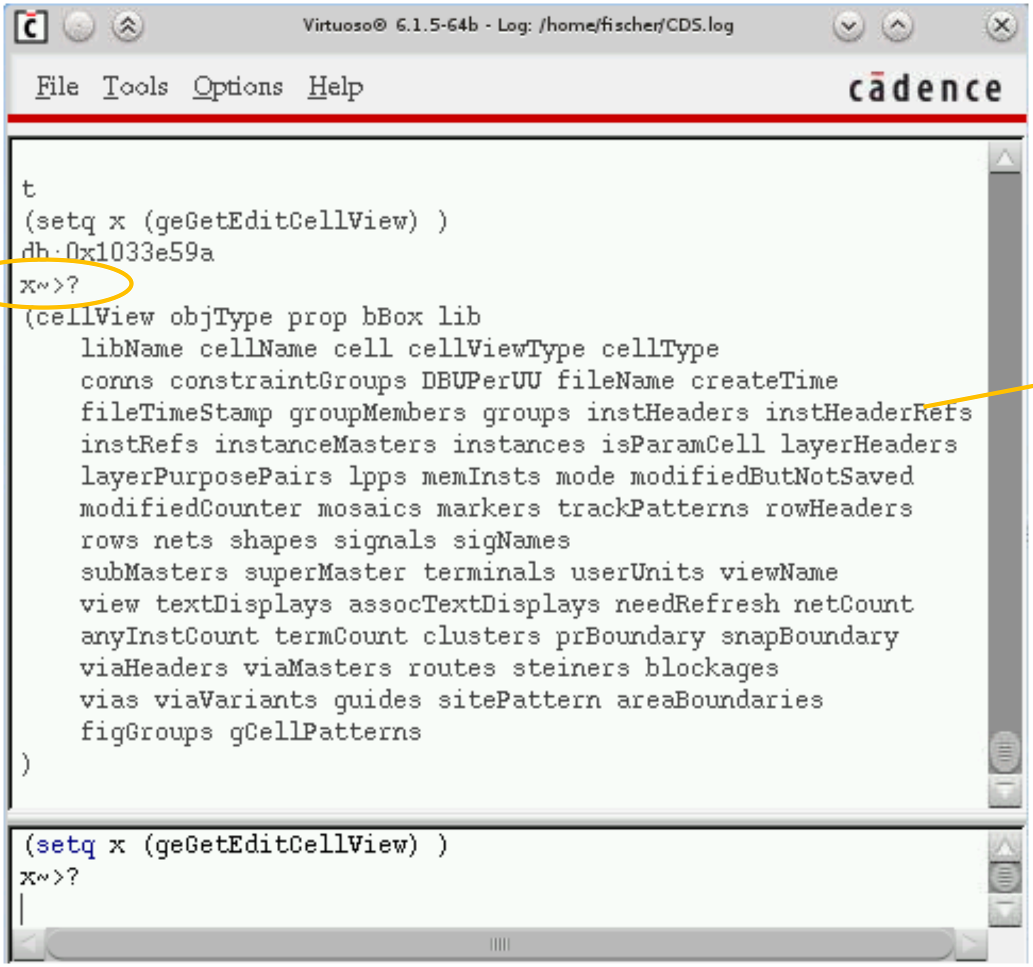
- Usefull: '`~>`' threads through lists, i.e. `list~>..` is possible!

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Getting access to an object (get the ID)

- With an open cell view (layout or schematic), the command **`(geGetEditCellView)`** gets the ID

commands of graphic editor (layout) mostly start with ge...



```
t
(setq x (geGetEditCellView) )
db:0x1033e59a
x~>?
(cellView objType prop bBox lib
    libName cellName cell cellViewType cellType
    conns constraintGroups DBUPerUU fileName createTime
    fileTimeStamp groupMembers groups instHeaders instHeaderRefs
    instRefs instanceMasters instances isParamCell layerHeaders
    layerPurposePairs lpps memInsts mode modifiedButNotSaved
    modifiedCounter mosaics markers trackPatterns rowHeaders
    rows nets shapes signals sigNames
    subMasters superMaster terminals userUnits viewName
    view textDisplays assocTextDisplays needRefresh netCount
    anyInstCount termCount clusters prBoundary snapBoundary
    viaHeaders viaMasters routes steiners blockages
    vias viaVariants guides sitePattern areaBoundaries
    figGroups gCellPatterns
)
```

```
(setq x (geGetEditCellView) )
x~>?
```

These are all the properties of the cell view

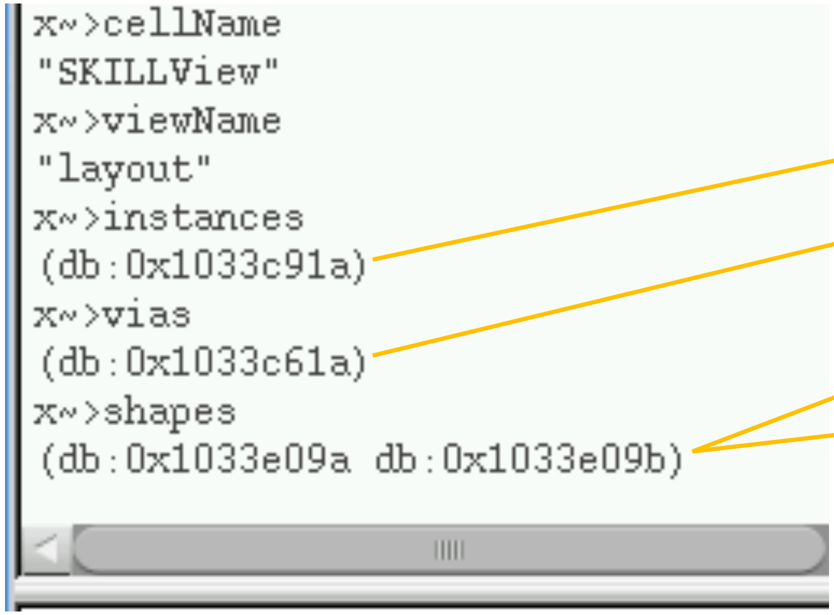
- Direct access via **`(dbOpenCellViewByType …)`**

dbGet

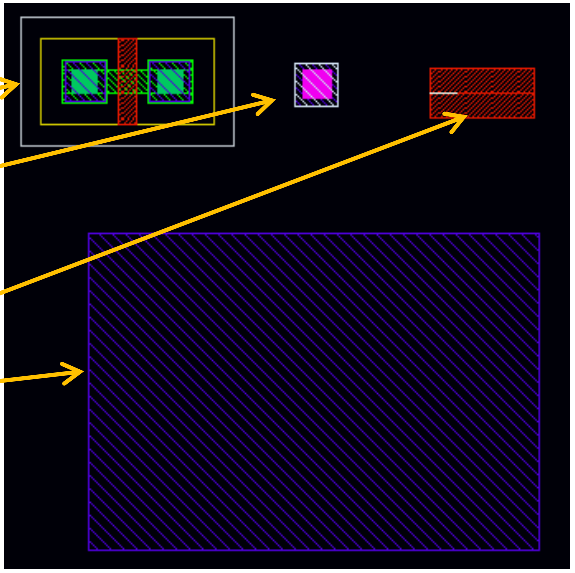
- Data base objects have (multiple) attributes (key / values - pairs)
  - For instance ‘objects’, ‘instances’, ‘bBox’,…
- The can be accessed by the dbGet function
  - See skdefref
- A short hand notation for attribute lookup is the `~>` operator

Looking at cell view properties

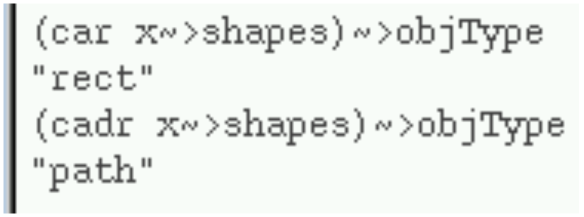
- Once we have a view ID, we can access the properties:



```
x~>cellName
"SKILLView"
x~>viewName
"layout"
x~>instances
(db:0x1033c91a)
x~>vias
(db:0x1033c61a)
x~>shapes
(db:0x1033e09a db:0x1033e09b)
```



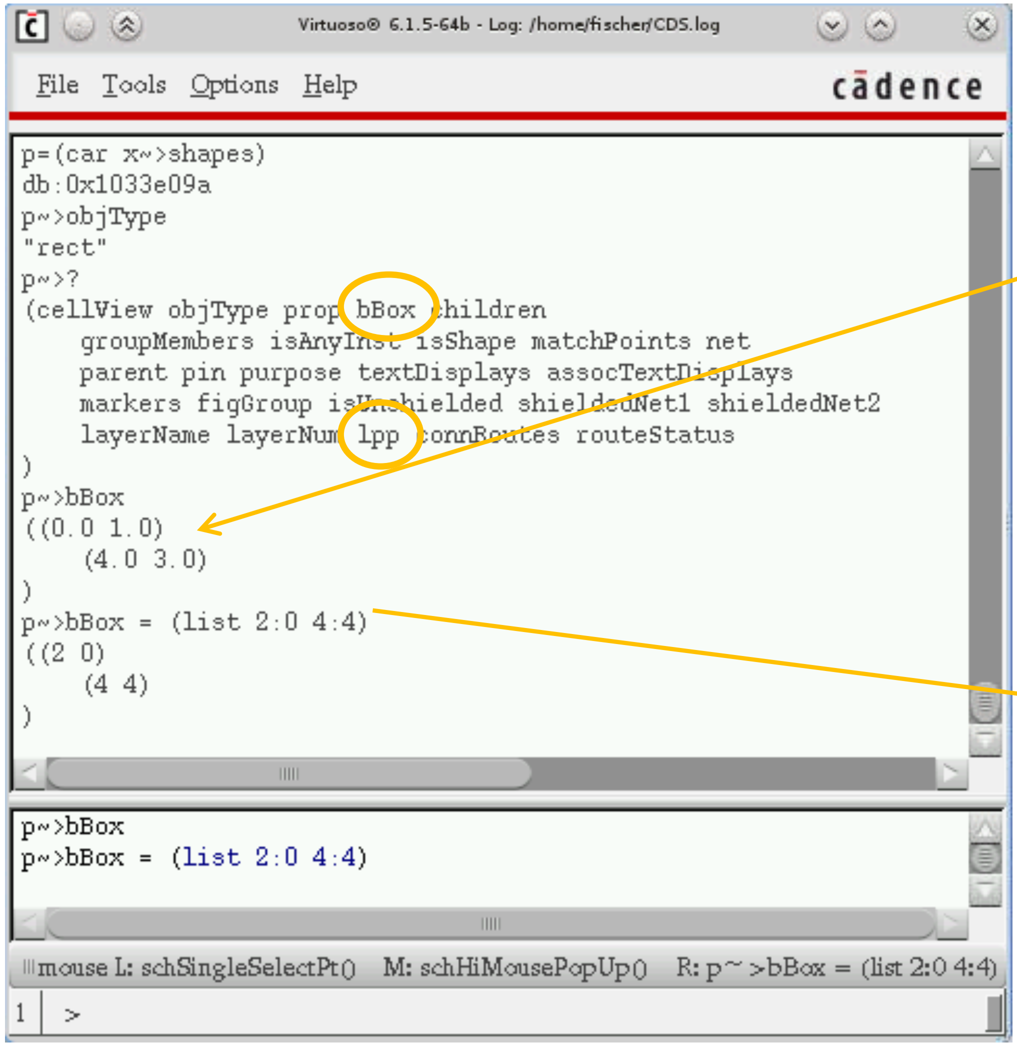
- The properties
  `instances,vias,shapes,layerPurposePairs (= lpp),..`
  are again *lists* of object **`ID`**s
- They can be studied further:



```
(car x~>shapes)~>objType
"rect"
(cadr x~>shapes)~>objType
"path"
```

Modifying Objects

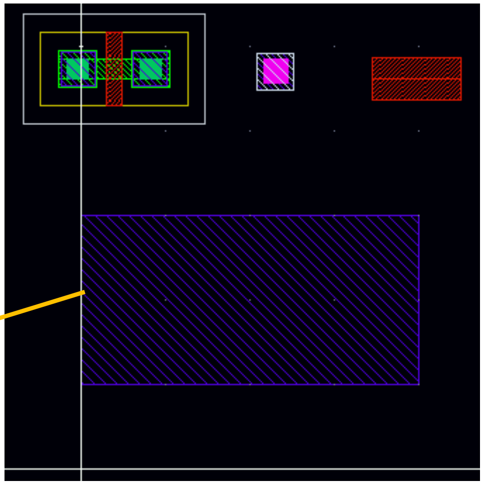
- The properties can be modified and affect the open view immediately:

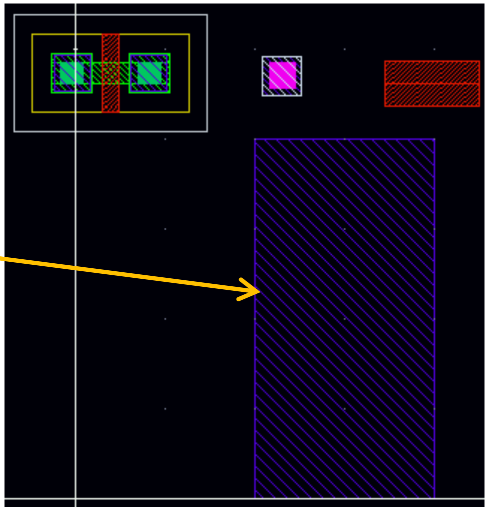


```
p=(car x~>shapes)
db:0x1033e09a
p~>objType
"rect"
p~>?
(cellView objType prop bBox children
    groupMembers isAnyInst isShape matchPoints net
    parent pin purpose textDisplays assocTextDisplays
    markers figGroup isUnshielded shieldedNet1 shieldedNet2
    layerName layerNum lpp connRoutes routeStatus
)
p~>bBox
((0.0 1.0)
    (4.0 3.0)
)
p~>bBox = (list 2:0 4:4)
((2 0)
    (4 4)
)
```

```
p~>bBox
p~>bBox = (list 2:0 4:4)
```

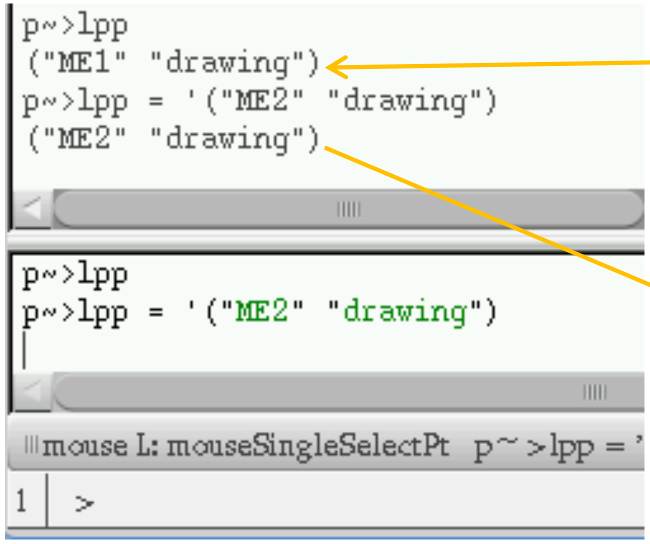
```
mouse L: schSingleSelectPt()    M: schHiMousePopUp()    R: p~>bBox = (list 2:0 4:4)
1  >
```





Modifying Objects - 2

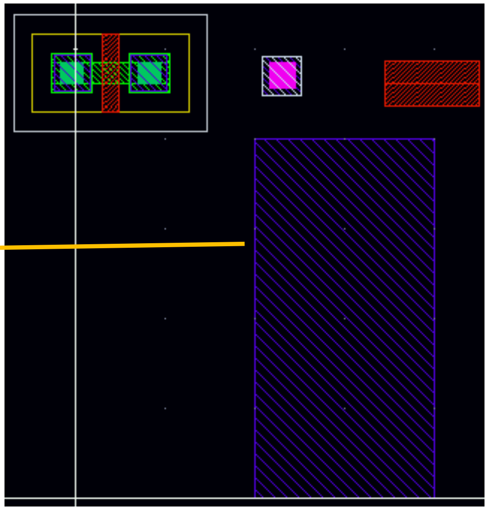
- The `layerPurposePairs` (`lpp`) defines the object layer
- It can be modified...

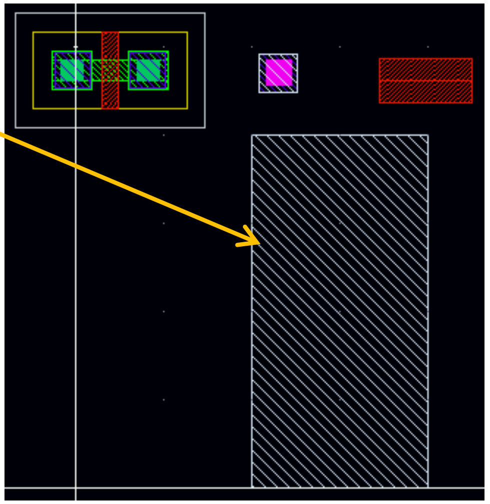


```
p~>lpp
("ME1" "drawing")
p~>lpp = '("ME2" "drawing")
("ME2" "drawing")
```

```
p~>lpp
p~>lpp = '("ME2" "drawing")
```

```
mouse L: mouseSingleSelectPt   p~>lpp = '
1   >
```



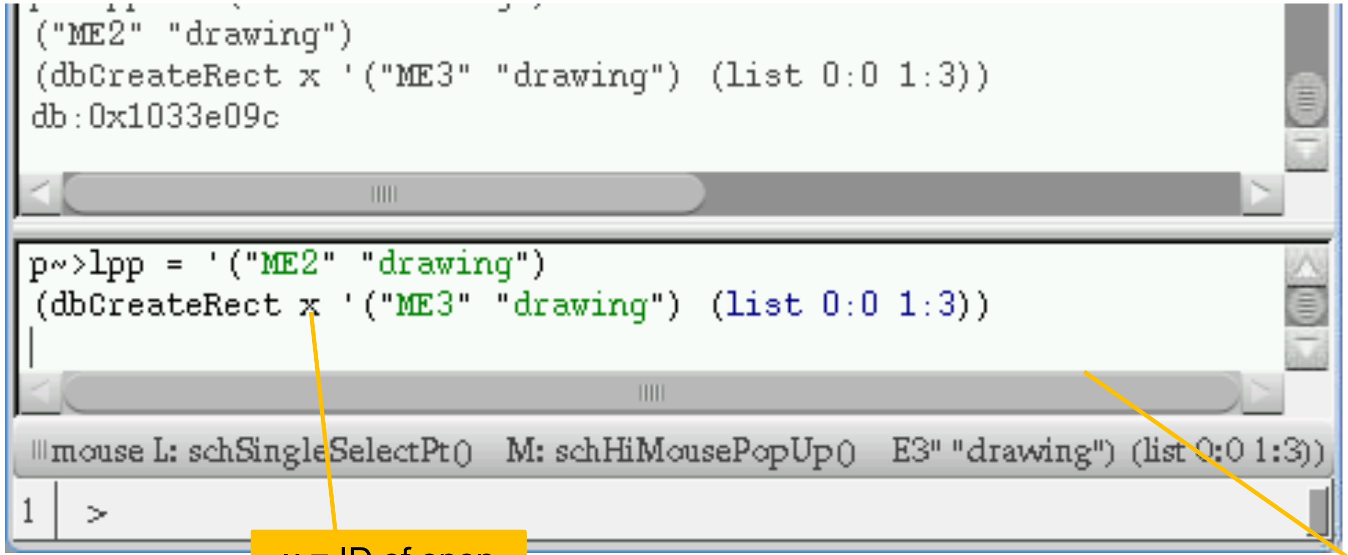


RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Creating New Objects

- There are many commands to create objects, see *skdfref*
- For instance, create a new rectangle with

`(dbCreateRect CellViewID lpp list(x:y x:y))`:

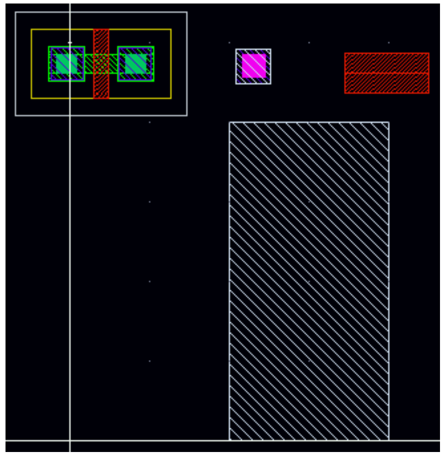


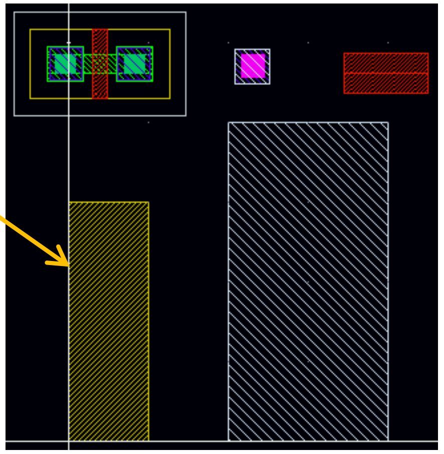
```
("ME2" "drawing")
(dbCreateRect x '("ME3" "drawing") (list 0:0 1:3))
db:0x1033e09c
```

```
p~>lpp = '("ME2" "drawing")
(dbCreateRect x '("ME3" "drawing") (list 0:0 1:3))
```

```
mouse L: schSingleSelectPt()   M: schHiMousePopUp()   E3" "drawing") (list 0:0 1:3))
1  >
```

x = ID of open cell view





Common Errors

- Common error:
  - No blank between a name and ‘`(`‘ in 'lisp' mode:
    `(setq a(plus a 3))`
  - This gives an error because `a(` is interpreted as function!!!

- Difference Integer / Float:
  - `(setq n 3)`
  - `(setq x n/2)` → x =1 !!! (integer division!)
  - `(setq x (float n)/2)` → x =1.5

Learning About Command Names

- When writing own command, the procedure names used by Cadence are sometimes difficult to find, despite the help files.
- Cadence tells you in the CIW which procedures are used by the *build-in commands* if you enable this under
- **`CIW->Options->Log Filter->\a`**

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Some Useful Applications of SKILL

Defining a Bindkey

- A bindkey (for layout editor) can be defined using the call
  `(hiSetBindKey "Layout" "key" "(function …params…)")`
  - first parameter is the tool ("Layout", "Schematic",..)
  - second is key (`"1"`, `"Ctrl v"`,…)
  - third is the function that will be called

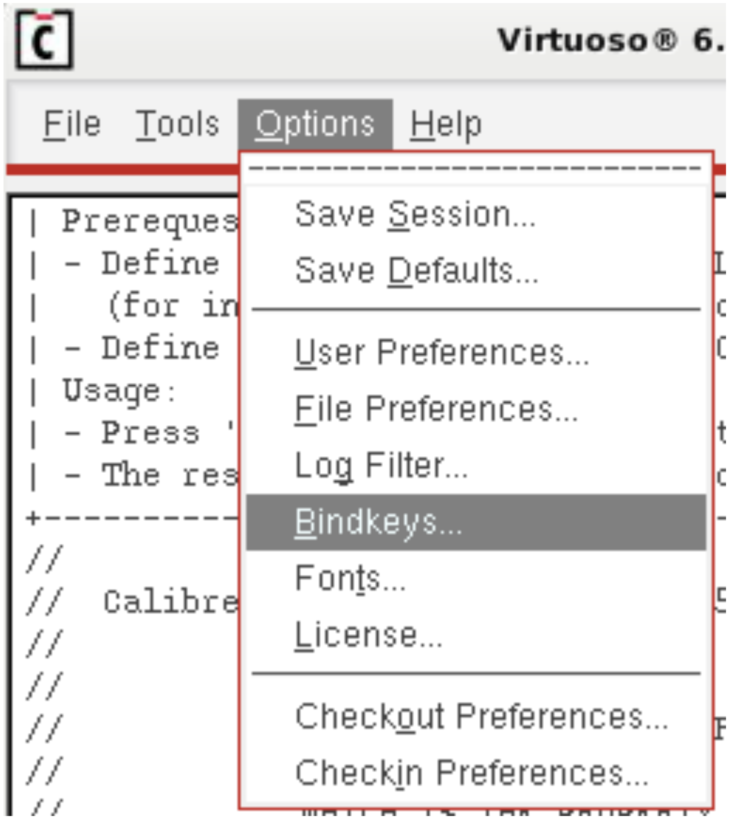
- For instance, you can set the snap grid with this procedure:

```
( procedure ( setSnapGrid snap )
  window = (hiGetCurrentWindow)
  window~>xSnapSpacing = snap
  window~>ySnapSpacing = snap
   (printf "Setting Snap Spacing to %.3f\n" snap)
)
```

- Install this with
  `(hiSetBindKey "Layout" "1" "(setSnapGrid 0.01)")`

Managing Bind Keys

- You can see all assignments to bind keys under **CIW → Options → Bindkeys**



Adding a User Menu and One Menu Item

```
(procedure (PrintNumberOfInstances)                    ; the procedure we install
   (setq inst (deGetEditCellView)~>instances )          ; (better use 'let'!)
   (printf "Found %d instances:\n" (length inst) )
   (foreach s inst (printf "%L\n" s->cellName) )
)
MyMenuShowObjects = ( hiCreateMenuItem            ; define an menu item
   ?name       'PrintNumberOfInstances            ; for later reference
   ?itemText   "Show # of Instances & types"      ; Text for menu item
   ?statusTip  "Show # inst & type"               ; shown in status bar
   ?callback   "PrintNumberOfInstances"
)
hiCreatePulldownMenu(                             ; define a menu
   'MyMenu                                        ; for later reference
   "MyFirstMenu"                                  ; text for menu (in bar)
   list( MyMenuShowObjects )                      ; all menu items in the menu
)
(procedure (InstallMenu args)                     ; installation routine which
   (hiInsertBannerMenu                            ; adds the menu to the 'Banner' bar
   args->window                                   ; the window
   MyMenu                                         ; name of menu to add
   (length (hiGetBannerMenus args->window))       ; menu position (0=left)
))
(deRegUserTriggers "maskLayout"                   ; tell cadence to call (InstallMenu)
  nil nil 'InstallMenu)                           ; whenever a layout is opened
```

Loading Things Automatically

- SKILL code can be executed automatically:
- At startup of Cadence, the file **`.cdsinit`** is executed.
  In this file, you can for instance define shortcuts
- When the layout/schematic editor is started, the files **`layEdit.menus`** or **`schematic.menus`** in directory **`menus`** (in the working dir) are executed
  - If you want to create your own menu, put them here.
- You can use some predefined menus (from SuS):
  - Create subdirectory **`menus`** (with command **`mkdir menus`**)
  - In this directory, create symbolic links with

    ```
    ln -s /shares/tools/SKILL/layEdit.menus layEdit.menus
    ```

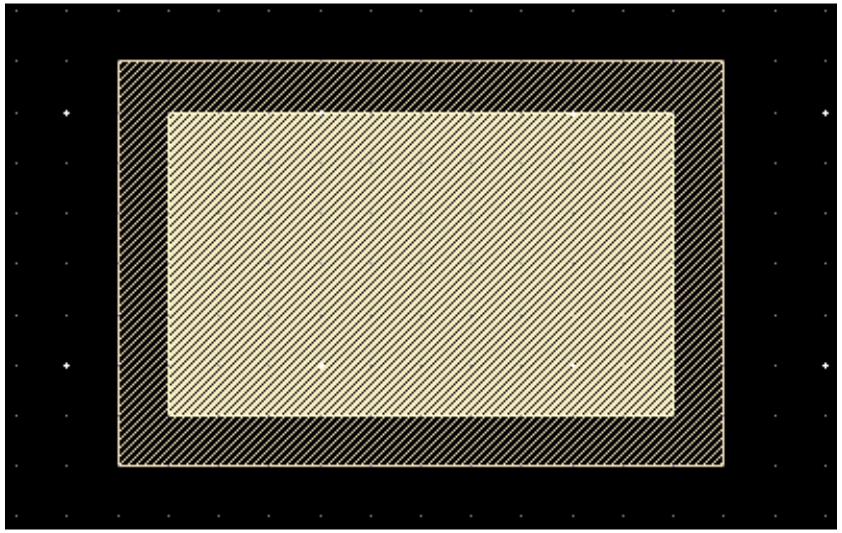
    and

    ```
    ln -s /shares/tools/SKILL/schematic.menus schematic.menus
    ```
  - You need to restart Cadence…

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Defining a Parameterized Cell

- You can create a fully new cell with Skill (layout, symbol,…)
- This cell can contain *parameters* which change its content
- A function defines how the cell looks in dependence of the parameter
- Example: A Pad with
  - an opening in layer "PAD" specified by two parameters (x,y)
  - metal6 around “PAD” with a 1µm extension
  - metal5 of similar size, but only if a (parameter) flag is set



RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

PCELL Definition (here: for Layout)

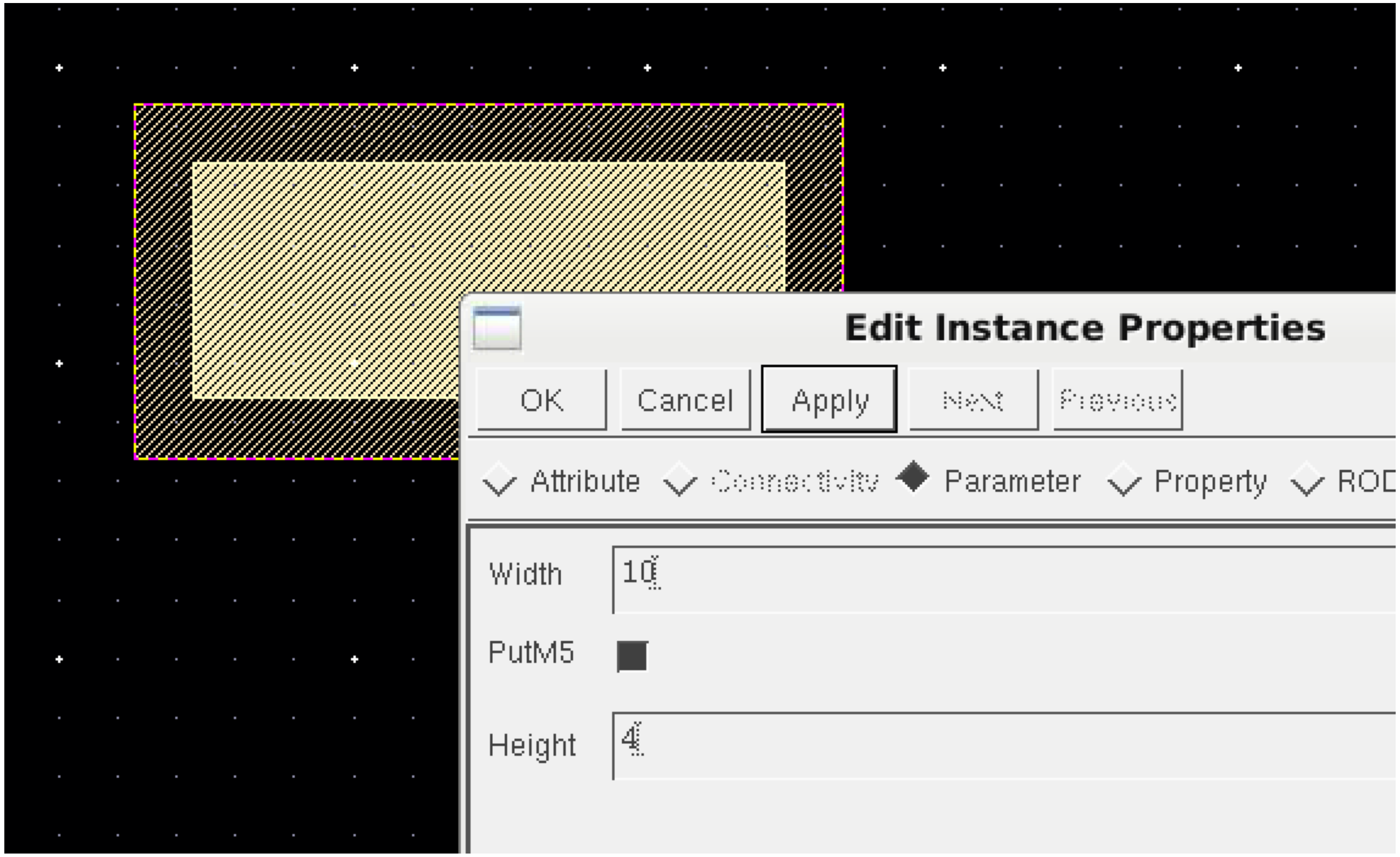
```
( pcDefinePCell
  (list (ddGetObj ″someLibrary") "TestPad" "maskLayout") ; the cell to create
  (
    (Width   "float"     10)   ; parameters with type and default
    (Height  "float"      6)   ; type can also be given in quotes “float”
    (PutM5   "Boolean"  nil)   ; a flag
  )
  ( let                     ; The code. Better use 'let' for local variables!
    (                                              ; local variables:
      (lppM5    '("ME5" "drawing"))                ; define a lpp for later usage
      (Overlap                    1.0 )            ; overlap of metals around PAD
     MetalShape                                    ; variable used internally later
    )
                                                   ; content starts here
    (setq MetalShape                               ; define larger metal shape for later
      ( list -Overlap:-Overlap Width+Overlap:Height+Overlap )
    )
    (dbCreateRect pcCellView '("PAD" "drawing")  (list 0:0 Width:Height))
    (dbCreateRect pcCellView '("ME6" "drawing")   MetalShape            )
    (if PutM5 (dbCreateRect pcCellView lppM5 MetalShape) )
  ) ; end let
  (return t)
)
```

Note: Make sure to use the correct type of "

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Instantiating the PCELL

- The 3 parameters can be changed



Width 10

PutM5 ■

Height 4

Note on PCELL

- Procedures should not be used in pcells, because they must be available at ***every*** evaluation of the pcell.
- Loading them via cdsinit is ok for interactive work, but not for some tools (like LVS with Assura)

- A solution is to load the procedures via the file libInit.il which must be located in the corresponding library directory.
  - (found by J. Klamroth here: https://community.cadence.com/cadence_technology_forums/f/custom-ic-skill/17440/how-to-use-procedure-inside-pcdefinepcell)

- Be careful with procedure names to avoid conflicts!
- (If remains to be checked that this works with all tools which evaluate pcells)

Some More Useful Commands…

- Open a view:

```
( dbOpenCellViewByType lib cell view type flag )
```

- List of objects marked with mouse:

```
( geGetSelectedSet )
```

- Get Coordinate of mouse:

```
( hiGetPoint (hiGetCurrentWindow) )
```

- Create a ruler:

```
( leCreateRuler view (list 0:0 0:10 10:10) )
```

- Move intersections of objects in lists to a cell on layer:

```
( dbLayerAnd cell layer list1 list2 )
```

(Extracting Schematic / Symbol / Layout to SKILL)

- To help in development of PCELLs, the content of open view can be converted into the SKILL code of a PCELL:
- From the open view, execute **`(pcHICompileToSkill)`** in the CIW
- Select schematic or symbol view, chose a SKILL file name