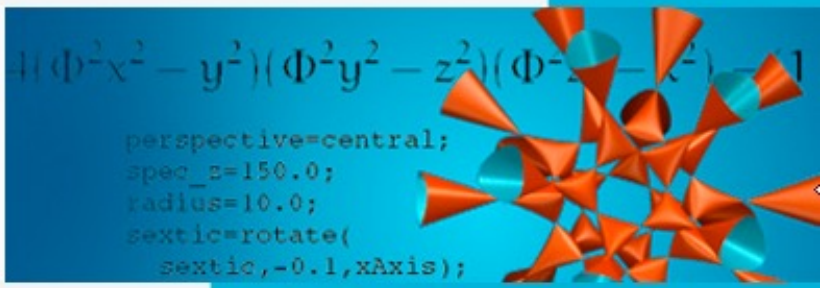
Philipps Universität Marburg

Declarative programming

Summer semester 2024

Prof. Christoph Bockisch, Steffen Dick
(Programming languages and tools)

Imke Gürtler, Daniel Hinkelmann, Aaron Schafberg, Stefan Störmer



[Script 2.4 - 2.5]

Conditional expressions

- Case-dependent functions
  - Various possible expressions
  - Selection based on condition (condition)
  - Condition: Value of an expression is true

Conditional expressions

```
(define (note points)
        (cond
                [(>= points 90) 1]
                [(>= points 80) 2]
                [(>= points 70) 3]
                [(>= points 60) 4]
                [(>= points 50) 5]
                [(< points 50) 6]))
> (note 95)
1
> (note 73)
3
> (note 24)
6
```

Meaning of conditional expressions

(cond

Keyword

[eCondition$_{,1}$ eResult$_{,1}$]

...

[eCondition$_{,n}$ eResult$_{,n}$])

- Informal meaning
  - Evaluation of *eCondition$_{,i\ in}$* sequence
  - For the first i with *eCondition$_{,i}$* true, *eResult$_{,i}$* is evaluated
  - The result is the value of the total expression

Errors with conditional expressions

- Expression for condition must return true or false (data type truth value)

```
>(cond [(+ 2 3) 4])
```

cond: question result is not true or false: 5

- One of the conditions must be true

```
>(cond [(< 5 3) 77]
       [(> 2 9) 88])
```

cond: all question results were false

Errors with conditional expressions

- Note the order of the conditions
- Put stronger conditions first

```
(define (note points)
        (cond
                [(< points 50) 6]
                [(>= points 50) 5]
                [(>= points 60) 4]
                [(>= points 70) 3]
                [(>= points 80) 2]
                [(>= points 90) 1]))
> (note 95)
```

Errors with conditional expressions

- Note the order of the conditions
- Put stronger conditions first

```
(define (note points)
        (cond
                [(< points 50) 6]
                [(>= points 50) 5]
                [(>= points 60) 4]
                [(>= points 70) 3]
                [(>= points 80) 2]
                [(>= points 90) 1]))
```

```
> (note 95)
5
```

(>= points 50) → true
However, this result is undesirable

Reduction with conditional expressions

- Reduction of an expression e

1. (PRIM) If e has the form (f v_1 ... v_n), f is a "**primitive**" function and the application of f to v_1 ... v_n has the value v,
then (f e_1 ... e_n) → v applies.

2. (FUN) If e has the form (f v_1 ... v_n), f is **not a primitive** function and the "context" contains the function definition of f:
(define (f x_1 ... x_n) e_{Body}),
then (f v_1 ... v_n) → $_{eNewBody}$ applies, whereby $_{eNewBody\ is\ created}$ from e_{Body} by replacing all x_i with v_i (i = 1 ... n).

3. (COND-false) If e has the form (cond [false e_1] [e_2 e_3] .... [e_{n-1} e_n]),
then (cond [false e_1] [e_2 e_3] .... [e_{n-1} e_n]) → (cond [e_2 e_3] .... [e_{n-1} e_n]) applies.

4. (COND-true) If e has the form (cond [true e_1] [e_2 e_3] .... [e_{n-1} e_n]),
then (cond [true e_1] [e_2 e_3] .... [e_{n-1} e_n]) → e_1

5. (KONG) If e has a sub-expression e_1 in an evaluation item*)
with e_1 → e_1 ',
then e → e' applies, whereby e' is generated from e by replacing e_1 with e_1 '.

Reduction with conditional expressions

- Reduction of an expression e

1. (PRIM) If e has the form (f v_1 ... v_n), f is a "**primitive**" function and the application of f to v_1 ... v_n has the value v,
then (f e_1 ... e_n)

2. (FUN) If e has the form
the function def
(define (f x_1 ... x
then (f v_1 ... v_n)
all x_i with v_i (i = 1 ... n).

the "context" contains

om e_{Body} by replacing

*) In an expression of the form
(cond [e_0 e_1] ... [e_{n-1} e_n])
is only e_0 an evaluation item.

3. (COND-false) If e has the form (cond [false e_1] [e_2 e_3] .... [e_{n-1} e_n]),
then (cond [false e_1] [e_2 e_3] .... [e_{n-1} e_n]) → (cond [e_2 e_3] .... [e_{n-1} e_n]) applies.

4. (COND-true) If e has the form (cond [true e_1] [e_2 e_3] .... [e_{n-1} e_n]),
then (cond [true e_1] [e_2 e_3] .... [e_{n-1} e_n]) → e_1

5. (KONG) If e has a sub-expression e_1 in an evaluation item*)
with e_1 → e_1 ',
then e → e' applies, where e' is generated from e by replacing e_1 with e_1 '.

Example: Reduction

(note 83)

→ (cond [(>= 83 90) 1] [(>= 83 80) 2] ...)

Rule FUN

```
(define (note points)
        (cond
                [(>= points 90) 1]
                [(>= points 80) 2]
                [(>= points 70) 3]
                [(>= points 60) 4]
                [(>= points 50) 5]
                [(< points 50) 6]))
```

Example: Reduction

(note 83)

→ (cond [(>= 83 90) 1] [(>= 83 80) 2] ...)

→ (cond [false 1] [(>= 83 80) 2] ...)

Rule KONG with rule PRIM:
(>= 83 90) → false

```
(define (note points)
        (cond
                [(>= points 90) 1]
                [(>= points 80) 2]
                [(>= points 70) 3]
                [(>= points 60) 4]
                [(>= points 50) 5]
                [(< points 50) 6]))
```

Example: Reduction

(note 83)

→ (cond [(>= 83 90) 1] [(>= 83 80) 2] ...)

→ (cond [false 1] [(>= 83 80) 2] ...)

Rule COND-false

→ (cond [(>= 83 80) 2] ...)

```
(define (note points)
        (cond
                [(>= points 90) 1]
                [(>= points 80) 2]
                [(>= points 70) 3]
                [(>= points 60) 4]
                [(>= points 50) 5]
                [(< points 50) 6]))
```

Example: Reduction

(note 83)

→ (cond [(>= 83 90) 1] [(>= 83 80) 2] ...)

→ (cond [false 1] [(>= 83 80) 2] ...)

→ (cond [(>= 83 80) 2] ...)

→ (cond [true 2] ...)

Rule KONG with rule PRIM
(>= 83 80) → true

```
(define (note points)
        (cond
                [(>= points 90) 1]
                [(>= points 80) 2]
                [(>= points 70) 3]
                [(>= points 60) 4]
                [(>= points 50) 5]
                [(< points 50) 6]))
```

Example: Reduction

(note 83)

→ (cond [(>= 83 90) 1] [(>= 83 80) 2] ...)

→ (cond [false 1] [(>= 83 80) 2] ...)

→ (cond [(>= 83 80) 2] ...)

→ (cond [true 2] ...)

→ 2

Rule COND-true

```
(define (note points)
        (cond
                [(>= points 90) 1]
                [(>= points 80) 2]
                [(>= points 70) 3]
                [(>= points 60) 4]
                [(>= points 50) 5]
                [(< points 50) 6]))
```

Example: Evaluation items

```
(define (rate amount total)
        (cond [(= total 0) 100]
              [true (* (/ amount total) 100)]))
>(rate 3 9)
33.3
>(rate 3 0)
100
```

If this were an evaluation item, the second printout would fail.

Task

1. Given a program:

```
(define (my-fun x)
      (cond [(and (>  x 10)  (<  x 20))  (-  22 x)]
                  [true (+  22 x)]))
```

What is a valid reduction of the following expression:

```
(my-fun 5)
```

a) 27

b)
```
(cond [(and (>  5 10)  (<  5 20))  (-  22 5)]  [true (+  22 5)])
```

c)
```
(cond [(and (>  5 10)  (<  5 20))  17]  [true (+  22 5)])
```

d)
```
(cond [(and (>  5 10)  (<  5 20))  (-  22 5)]  [true 27])
```

e)
```
(cond [(and (>  5 10)  true)  (-  22 5)]  [true (+  22 5)])
```

f)
```
(cond [true (+  22 5)])
```

Live Vote

https://ilias.uni-marburg.de/vote/IQXW

Syntactic sugar

- Some program constructs appear frequently
- Independent of the project
- To avoid redundancy
  - "Syntactic sugar"
  - Not a new language construct!
  - Simplified notation equivalent to existing language construct

Standard case for conditional expressions

- One case must always apply
- Can be achieved by condition true
- Disadvantage: unclear whether "standard case" is meant or expression random/false true
- Syntactic sugar: Keyword else

```
(define (note points)
        (cond
                [(>= points 90) 1]
                [(>= points 80) 2]
                [(>= points 70) 3]
                [(>= points 60) 4]
                [(>= points 50) 5]
                [else 6]))
```

Standard case for conditional expressions

- Advantage of else
  - Intention of the programmer explicit
  - Can be used by programming language
  - Standard case is only used if all other cases are not fulfilled
  - else can only be used in the latter case

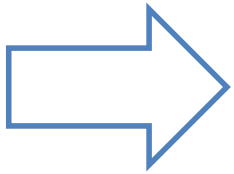
```
>(cond [(> 3 2) 5]
       [else 7]
       [(< 2 1) 13])
```

cond: found an else clause that isn't the last clause in its cond expression

Transformation for others

- Syntactic sugar adds nothing new
- Abbreviation only
- Therefore no reduction rules, but transformation

(cond $[e_0\ e\ ]_1$
$[e_2\ e\ ]_3$
...
[else e_n])



(cond $[e_0\ e\ ]_1$
$[e_2\ e\ ]_3$
...
[true e_n])

else becomes true.

Only one condition

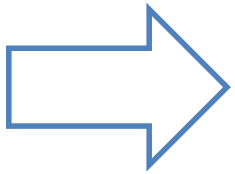
- With cond expressions, the evaluation depends on any number of conditions
- Often only one condition is relevant ("either one or the other")
- Syntactic sugar: if
- Advantage: better readability

```
(define (frost? temperature)
        (if (< temperature 0) true false))
> (frost? -5)
true
```

if-expressions

- Transformation

(if *eCondition* *e* *e*)*Then Else*

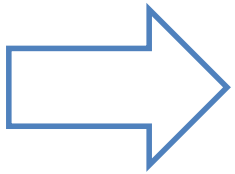


(cond [*eCondition* *e*]*Then*
[else e_{Else}])

if and cond

- if and cond are equivalent
- This means that every cond expression can also be transformed into an if expression

(cond [e_0 e]$_1$
[e_2 e]$_3$
...
[e e]$_{n-2\ n-1}$
[else e_n])



(if e_0 e_1
(if e_2 e_3
(...
(if e_{n-2} e_{n-1} e_n) ...)))

Several conditions lead to nested if expressions. A cond expression is easier to read here.

Definition of constants

- Previously: user-defined functions
- We can also define constants

```
> (define B 42)
> B
42
```

From here on, B can be used as an expression and corresponds to the value 42.

- Constant definitions do not result in a value
- However, may be specified in the interaction window

Constant definitions

(define ConstantName BodyExpression)

- define
  - Keyword introduces constant and function definition
- ConctantName
  - Name via which the constant can be used
- BodyExpression
  - Expression that determines the value of the constant
  - Evaluated as soon as the constant is defined

Constant definitions

- When reading in the constant definition, DrRacket immediately evaluates the body expression
- This value is used later for all occurrences of the constants
- Is that allowed?

```
(define A (+ B 1))
(define B 42)
> A
```

Constant definitions

- When reading in the constant definition, DrRacket immediately evaluates the body expression
- This value is used later for all occurrences of the constants
- Is that allowed?

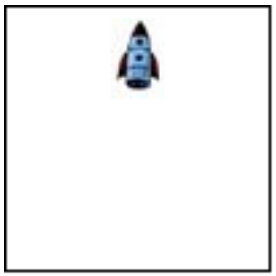
```
(define A (+ B 1))
(define B 42)
> A
```

No!

B is used here before its definition

Magic Numbers

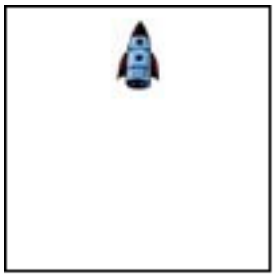
- >(place-image  20 (empty-scene 100 100))



Why 50? Does it always have to be 50?

Magic Numbers

- >(place-image  20 (empty-scene 100 100))



Why 50? Does it always have to be 50?

The number 50 has a certain meaning here!
The horizontal center of the scene

- The meaning of a number cannot be seen
- But it is there, which is why we speak of "magic numbers"

Magic Numbers

- >(place-image  20 (empty-scene 150 100))

Width is changed. What does this mean?

Magic Numbers

- >(place-image  20 (empty-scene 150 100))

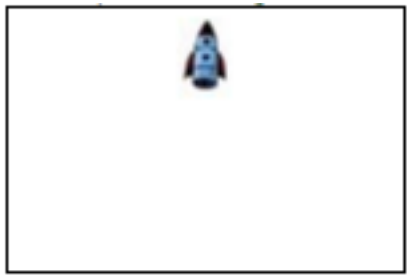
The horizontal center depends on the width.

Width is changed. What does this mean?

Magic Numbers

- Avoid magic numbers with constants

- (define WIDTH 150)
- (define CENTER (/ WIDTH 2))
- >(place-image ENTER 20 (empty-scene WIDTH 100))



Meaning of function and constant definitions

- Function definition
  - Function names, parameters and body expression are registered
  - **No evaluation**
- Constant definition
  - Body expression is **evaluated**
  - Constant name and value of the body expression are registered

Reduction with constants

- Reduction of an expression *e*

1. (PRIM) If *e* has the form (f v_1 ... v_n), f is a "**primitive**" function and the application of f to v_1 ... v_n has the value *v*,
then (f e_1 ... e_n) → *v* applies.

2. (FUN) If *e* has the form (f v_1 ... v_n), f is **not a primitive** function and the "context" contains the function definition of f:
(define (f x_1 ... x_n) e_{Body}),
then (f v_1 ... v_n) → $_{eNewBody}$ applies, whereby $_{eNewBody\ is\ created}$ from e_{Body} by replacing all x_i with v_i (i = 1 ... n).

3. (COND-false) If *e* has the form (cond [false e_1] [e_2 e_3] .... [e_{n-1} e_n]),
then (cond [false e_1] [e_2 e_3] .... [e_{n-1} e_n]) → (cond [e_2 e_3] .... [e_{n-1} e_n]) applies.

4. (COND-true) If *e* has the form (cond [true e_1] [e_2 e_3] .... [e_{n-1} e_n]),
then (cond [true e_1] [e_2 e_3] .... [e_{n-1} e_n]) → e_1

5. (CONST) If *e* is the symbol *c* and the "context" contains the constant definition of c:
(define *c* e_{Body}) with e_{Body} →* *v*,
then *e* → *v*

6. (KONG) If *e* has a sub-expression e_1 in an evaluation position with e_1 → e_1 ',
then *e* → *e'* applies, whereby *e'* is generated from *e* by replacing e_1 with e_1 '.

Meaning of a program

- Program
  - Sequence of expressions, constant definitions and function definitions ("program elements")
  - Any order

According to convention:
Definitions before expressions

- Context
  - Regarding a program element
  - Set of all function and constant definitions up to the program element

Meaning of a program

- Program is evaluated from left to right and from top to bottom
  If the next program element is ...
  - ... an expression, it is evaluated according to t reduction rules.
  - ... a function definition, then this is added to the context
  - ... a constant definition, the body expression is evaluated according to the reduction rules and the constant with this value is added to the context

No statement about the order of reductions but about the evaluation of expressions in the sequence.

Tasks

What is the result of the following programs?

1.
```
(define (g x) (* F (f x)))
(define (f x) (+ x 1))
(define F 4)
(g 2)
```
   a) 12
   b) 5
   c) A mistake

2.
```
(define A 42)
A
```
   a) 42
   b) A mistake

3.
```
(define A (* B 2))
(define B 3)
(A)
```
   a) 6
   b) 0
   c) A mistake