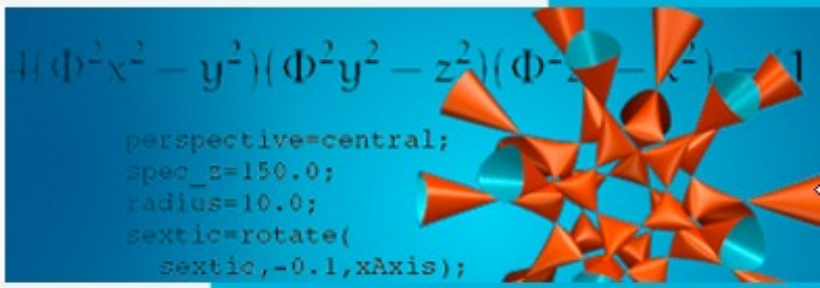
Philipps Universität Marburg

Declarative programming

Summer semester 2024

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

Imke Gürtler, Daniel Hinkelmann, Aaron Schafberg, Stefan Störmer



[1.1 - 1.3, 1.5 - 2.1]

Literals

- Literals:
  - Values that appear directly in the program code
  - "Atomic expressions"
- Previously: Number literals
- Other types of literals
  - Text
  - Truth values
  - Pictures

Data types

- All values can be divided into "data types" (sorts)
- The same functions can be applied to values of the same data type (the result may be different)
- For example, numbers can be added, texts cannot

Data types - Constructors

- Values can be generated using literals, for example

- 3
- 5.3
- "Declarative programming"
- true

Data types - Functions

- Functions expect arguments that return values of certain data types
- A "function is defined on a data type"

- Functions on the String (text) data type:

```
> (string-append "Declarative " "Programming")
"Declarative programming"
```

- Etc.
- → "Arithmetic of strings"

- List of predefined functions:
  https://docs.racket-lang.org/htdp-langs/beginner.html

Functions

- So far
  - Binary operations
  - Data type of arguments and result equal

- Functions can have different numbers of arguments

```
> (+ 2 3 4)
9
```

- Functions can expect and return values of different data types

```
> (+ (string-length "Programming languages") 5)
26
```

21

Functions

- Functions can expect arguments of different data types

```
> (replicate 3 "hi")
"hihihi"
```

- Functions can convert data types

```
> (number->string 42)
"42"
> (string->number "42")
42
```

Representations

- Number literal
  - 42
- Stringliteral
  - "42"
- Expression
  - (+ 21 21)
- Stringliteral
  - "(+ 21 21)"

Similar, but not compatible!

```
> (+ "42" 1)
+: expects a number as 1st argument, given "42"
```

Data type: Truth values

- Two literals: true, false.
- Propositional logic functions, e.g.

```
> (and true true)
true
> (and true false)
false
```

- Comparisons, e.g.

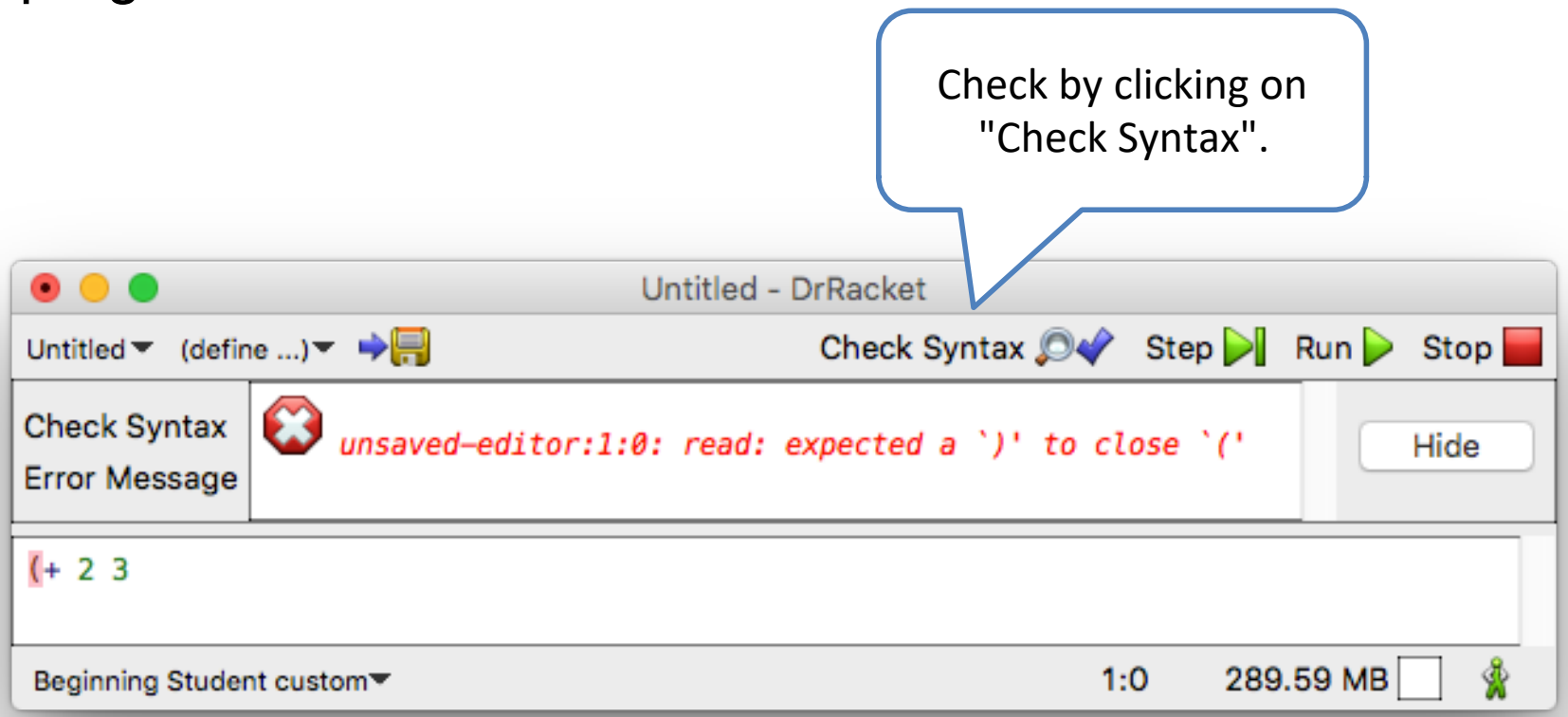
```
> (> 10 9)
true
> (= 42 9)
false
> (string=? "hello" "world")
false
```

Syntax

- Programming languages have a grammar (syntax)
- The grammar specifies the structure a program must have
- Grammar of a Dr Racket Beginning Student Language (BSL) program
  - Program: Sequence of expressions
  - Expression: Literal or function call
  - Literal: number, image, truth value, string
  - Function call: **(f a1 a2 ...)**, where f is the function name and a1, a2, ... are expressions

Syntax error

- The syntax can be checked without executing the program



Other errors

- Not all syntactically correct programs have a meaning
- This means that not all syntactically correct expressions can be evaluated
- Errors that are not syntax errors occur at runtime

```
> (number->string "asdf")
number->string: expects a number, given "asdf"
> (string-length "asdf" "fdsa")
string-length: expects only 1 argument, but found 2
```

Type error: actual type and expected type do not match.

Other errors

```
> (/ 1 0)
/: division by zero
```

Function is not defined for all arguments.

Error codes

- Errors can also be suppressed

```
> (string->number "asdf")
false
```

- "asdf" does not represent a number
  - No runtime error occurs
  - Instead, false is returned

- The program itself can decide how to deal with incorrect conversions

Timing of errors

- Early errors are easier to rectify
- Point in time:
  Syntax error < runtime error < error codes
- Late errors are less restrictive

Meaning of expressions

- A BSL program consists of expressions
- Description of any BSL programs using variables (in italics)
- Naming convention
  - Starting with *e*: any expression (*e*, e_1 , *e*′, ...)
  - Starting with *v*: Value, e.g. a number, a string, etc. (*v*, v_1 , *v*′, ...)

Algorithm for determining the meaning

1. Given an expression *e*, its meaning is
   a. If *e* is already a value, then this value is its meaning
   b. If *e* has the form (f e_1 ... e_n), *e* is evaluated as follows
      i. Are e_1 ... e_n are already values v_1 ... v_n and f is defined on v_1 ... v_n is defined,
      then the value of *e is* the application of f to v_1 ... v_n
      ii. Are e_1 ... e_n are already values v_1 ... v_n but f is **not defined** on v_1 ... v_n is defined,
      then the evaluation is aborted with a runtime error
      iii. Otherwise, determine the value v_i for each expression e_i not yet evaluated by applying step 1 and replacing e_i with v_i . Then carry out step 1.b (now either 1.b.i or 1.b.ii is applicable)

Gradual reduction

- $e \rightarrow e'$: e can be reduced to e' in one step (reduction of an expression e) if one of these two rules is applicable
  1. If e has the form (f v_1 ... v_n) and the application of f to v_1 ... v_n has the value v,
     then (f v_1 ... v_n) $\rightarrow$ v applies.
  2. If e has a sub-expression e_1 in an evaluation item
     with $e_1 \rightarrow e_1$ ',
     then $e \rightarrow e'$ applies, whereby e' is generated from e by replacing e_1 with e_1 '.

- Rule 2 is called the "congruence rule"
- Values can no longer be reduced

Gradual reduction

- $e \rightarrow e'$: e can be reduced to e' in one step (reduction of an expression e) if one of these two rules is applicable
  1. If e has the form (f v_1 ... v_n) and the application of f to v_1 ... v_n has the value v,
     then (f v_1 ... v_n) $\rightarrow$ v applies.
  2. If e has a sub-expression e_1 in an evaluation item with $e_1 \rightarrow e_1'$,
     then $e \rightarrow e'$ applies, where e' is generated from e by replacing e_1 with e_1'.
- Rule 2 is called the
- Values can no longer be reduced

So far, all items are also evaluation items.

Conventions

- $e_1 \rightarrow e_2 \rightarrow e_3$ means: $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$
- $e \rightarrow^* e'$ means:
  - There is an $n >= 0$ and e_1 , ..., e_n , so that $e \rightarrow e_1 \rightarrow ... \rightarrow e_n \rightarrow e'$
- The following always applies: $e \rightarrow^* e$
- $\rightarrow^*$ is called the "reflexive-transitive closure of $\rightarrow$" (i.e. all indirectly reachable reductions)

Examples

- (+ 1 1) → 2

- The congruence rule can be applied in any order:
- (+ (* 2 3) (* 4 5)) → (+ 6 (* 4 5)) → (+ 6 20) → 26
- (+ (* 2 3) (* 4 5)) → (+ (* 2 3) 20) → (+ 6 20) → 26

Confluence

- The order of application of the congruence rule does not influence the result of the evaluation!

- Confluence rule:
  - Given $e_1 \rightarrow e_2$ and $e_1 \rightarrow e_3$,
    then: there is an e_4 such that $e_2 \rightarrow^* e_4$ and $e_3 \rightarrow^* e_4$

Equivalence

- The following applies: $e_1 \equiv e_2$ (e_1 is equivalent to e),$_2$ if an expression e exists such that $e_1 \rightarrow^* e$ and $e_2 \rightarrow^* e$
- Examples

  (+ 1 1) ≡ 2

  (+ (* 2 3) 20) ≡ (+ 6 (* 4 5)) ≡ 26
- The meaning of a program therefore does not change if we replace sub-expressions with equivalent expressions

Live Vote

https://ilias.uni-marburg.de/vote/Z77T

Tasks

1. What results from the reduction of:
   `(+ (- 5 3) (/ (+ 3 1) (- 4 (* 2 2))))`
   a) 2
   b) 8
   c) A mistake
2. What results from the reduction of:
   `(* (- (sqrt (+ 41 (* 2 4)))) 5) (* 3 7))`
   a) 42
   b) -42
   c) A mistake
3. To which expressions can the following expression be reduced?
   `(* (+ 5 3) (/ (+ 4 2) (- 8 (* 2 3))))`
   a) `(* (+ 5 3) (/ (+ 2 4) (- 8 (* 2 3))))`
   b) `(* 8 (/ (+ 4 2) (- 8 (* 2 3))))`
   c) `(* (+ 5 3) (/ (+ 4 2) 2))`
   d) `(* (+ 5 3) (/ (+ 4 2) (- 8 6)))`