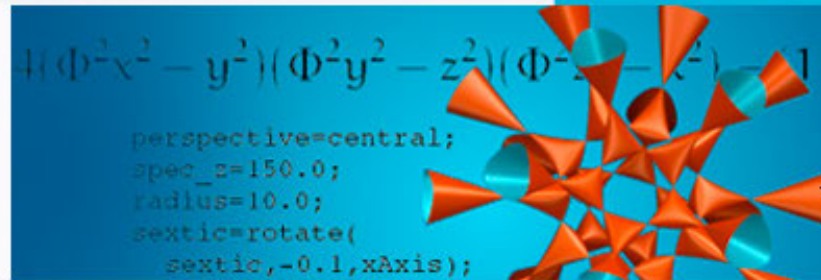


# Deklarative Programmierung

Sommersemester 2017

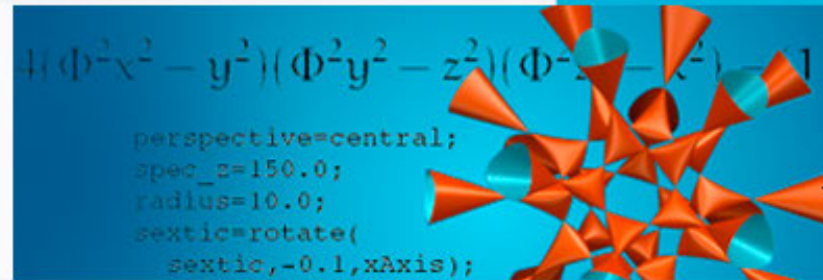
Prof. Christoph Bockisch  
(Programmiersprachen und –werkzeuge)  
Steffen Dick, Alexander Bille, Johannes Frankenau,  
Patrick Frömel, Niclas Schmidt, Jonas Stettin,  
Robert Tran, Julian Velten



[Skript 1 – 2.1]



# Organisation



# Organisation

- Vorlesung
  - Dienstags 16:15 – 17:45 in HC 8, HS B
  - Mittwochs 16:15 – 17:45 in HC 8, HS B
- Präsenzübungen
  - Donnerstags 10:00 - 12:00, in MZ 6, SR XI  
Steffen Dick
  - Donnerstags 10:00 - 12:00, in MZ 6, SR XIII  
Niclas Schmidt
  - Donnerstags 12:00 - 14:00, in MZ 6, HS VI  
Alexander Bille
  - Donnerstags 14:00 - 16:00, in MZ 6, SR XI  
Jonas Stettin
  - Donnerstags 16:00 - 18:00, in MZ 6, SR XI  
Patrick Frömel
  - Freitags 10:00 - 12:00, in MZ 6, SR XI  
Johannes Frankenau
  - Freitags 10:00 - 12:00, in MZ 6, SR XIII  
Robert Tran
  - Freitags 12:00 - 14:00, in Stadtmitte? (Raum TODO)  
Julian Velten
- Klausur
  - 18. Juli 2018, 13:00 – 16:00 in HC 8, HS A (Erstklausur)
  - 04. Oktober 2018, 13:00 – 16:00 in HC 8, HS A (Zweitklausur)

# Zeitplan

	1. Woche 17.04.	2. Woche 16.04.	3. Woche 23.04.	4. Woche 30.04.	5. Woche 07.05.	6. Woche 14.05.	7. Woche 21.05.
Mo	–	HÜ	HÜ	HÜ	HÜ	–	HÜ
Di	VL	VL	VL	–	VL	VL	VL
Mi	VL	VL	VL	VL	VL	–	–
Do	–	UE	UE	UE	–	UE	UE
Fr	–	UE	UE	UE	–	UE	UE

Tag der Arbeit

Christi Himmelfahrt

Sprt Dies

Dies Academicus

	8. Woche 28.05.	9. Woche 04.06.	10. Woche 11.06.	11. Woche 18.06.	12. Woche 25.06.	13. Woche 02.07.	14. Woche 09.07.
Mo	HÜ	–	HÜ	HÜ	HÜ	HÜ	–
Di	VL	VL	VL	VL	VL	VL	PQ
Mi	VL	PQ	VL	VL	VL	VL	Q/A
Do	–	UE	UE	UE	UE	UE	UE
Fr	–	UE	UE	UE	UE	UE	UE

Fronleichnam

15. Woche: Klausur am Mittwoch

# ILIAS

- Materialien via ILIAS
  - Script, Folien, Präsenz- und Hausübungszettel
  - Übungsgruppen (Tutorien + Kleingruppen)
  - Einreichung Hausübungen und Feedback
- Beitreten zur Vorlesung:  
Magazin → FB 12 → Informatik → Softwaretechnologie →  
SoSe 2018 → Bockisch: VL Deklarative Programmierung
- Beitreten zur Übungsgruppe
  - Ab 12.04. 20 Uhr möglich
  - Änderungen möglich bis 18.04. (danach nur noch durch Veranstalter)

# Hausübungen

- Werden jeweils Montags (12:00h) in ILIAS zur Verfügung gestellt
- Zu bearbeiten innerhalb von 7 Tagen  
(Frist jeweils folgender Montag 11:59h)
- Einreichung in Kleingruppen (2 – 3 Personen)  
**Ausnahme: 1. Übungsblatt muss einzeln bearbeitet und eingereicht werden.**
- 10 Übungsblätter à 16 Punkte: Maximum = 160 Punkte
- Klausurzulassung:
  - 50% der Punkte + Bearbeitung aller Übungsblätter (max. 2 Auslassungen)
  - Für Studierende des Lehramt für Gymnasien gilt: 40% der Punkte + fachdidaktische Zusatzleistung (melden Sie sich bitte bei mir: [bockisch@mathematik.uni-marburg.de](mailto:bockisch@mathematik.uni-marburg.de))
- Bonuspunkt:
  - Bei Erreichen von 80% der Punkte: Bonus von 1 Notenschritt auf die Klausur
  - Nur, wenn die Klausur auch ohne Bonuspunkt bestanden ist
  - Für Klausuren, die in 2018 geschrieben werden
  - Gilt gleichermaßen für Lehramt Studenten
- **Anleitung + Anweisungen in ILIAS**

# PubQuiz

- In der Vorlesung
- Ca. 8 Aufgaben werden für jeweils wenige Minuten angezeigt
- Arbeit in Kleingruppen (ca. 3 Studierende)
  - Ergebnisse der Aufgaben werden in Formular eingetragen
- Bewertung
  - Formulare werden anschließend bewertet und die Aufgaben besprochen
- “Preis”: zusätzliche Übungspunkte
  - Abhängig von der maximal erreichten Anzahl Punkte ( $P_{\max}$ )
  - Bei  $P_{\max}$  korrekten Antworten: 4 Bonuspunkte
  - Bei  $P_{\max} - 1$  korrekten Antworten: 3 Bonuspunkte
  - etc.
  - Maximal 1 Bonuspunkt pro korrekter Antwort (d.h. bei  $P_{\max} = 3$  gibt es im Höchstfall 3 Bonuspunkte)

# Lernziele

- Erlernen einer deklarativen Programmiersprache
- Erkennen und Anwenden von Abstraktionen
- Verstehen und Erkennen von sprachübergreifenden Konzepten und deren Einsatzgebieten
- Formale Festlegung von Semantik von Programmiersprachen
- Wissenschaftliches Arbeiten und Kommunikationsfähigkeiten



# Themen

- Programmieren mit Ausdrücken
- Programmieren mit Funktionsdefinitionen
- Konditionale Ausdrücke
- Entwurf von Funktionsdefinitionen
- Algebraische Datentypen
- Beweis von Programmeigenschaften
- Abstraktion über Funktionen und Typen
- Scoping
- Pattern Matching
- Generative Rekursion
- Akkumulatoren
- Logik-Programmierung

# Organisation

- Script
  - Basiert auf Skript von Prof. Klaus Ostermann
  - Basiert auf "How to Design Programs" von M. Felleisen, R.B. Findler, M. Flatt und S. Krishnamurthi
- Programmiersprache: DrRacket (Version 6.12)

# DrRacket

- Lisp Dialekt
- Verschiedene Sprach-niveaus auswählbar
- Installation: <http://racket-lang.org>
- Einstellungen
  - Sprache -> Sprache auswählen: "How to Design Programs – Anfänger"
  - Sprache -> Teachpack hinzufügen: "universe" und "2htdp/image"
- Benutzung
  - Interaktionsbereich: untere Bildschirmhälfte
  - Ausdruck eingeben, mit Enter bestätigen → Ergebnis wird in nächster Zeile ausgegeben

# DrRacket Installation

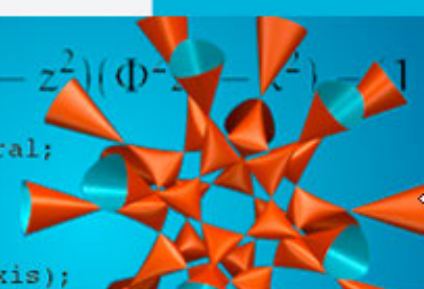
Installieren Sie DrRacket.

Bringen Sie Ihr Notebook zu den Übungen mit.

Bei Problemen mit der Installation können wir Sie während der Vorlesung (jetzt oder nächste Woche) oder während der Präsenzübungen unterstützen.



# Einleitung

A 3D visualization of a complex mathematical surface, rendered in orange and red, showing a series of interconnected, curved, and pointed structures. The surface is composed of many small, triangular facets, giving it a faceted appearance. It is set against a dark blue background.

```
4(\Phi^2x^2 - y^2)(\Phi^2y^2 - z^2)(\Phi^2z^2 - x^2) - 1  
perspective=central;  
spec_r=150.0;  
radius=10.0;  
sextic=rotate(  
    sextic,-0.1,xAxis);
```

# Algorithmen

- Präzise Definition der Schritte zum Lösen eines Problems
- Dasselbe Problem kann durch verschiedene Algorithmen lösbar sein
- Programmieren von Algorithmen ermöglicht Ausführung durch Computer
- Zur einfacheren Verständlichkeit verwenden wir Abstraktionen

# Arithmetik

- Programmiersprachen enthalten bereits definierte Algorithmen für wiederkehrende Aufgaben
  - Auswerten von Ausdrücken
  - Anwenden von Operationen etc.

# Einfache Auswertungen

Wir wollen diesen Ausdruck auswerten:

$$3 + 4 * 5$$

Was kann schief gehen?



# Einfache Auswertungen

Man kann fälschlich "+" zuerst auswerten und den Ausdruck als "7 \* 5" lesen.

$$3 + 4 * 5$$

Wie können wir solche Fehler vermeiden?

# Einfache Auswertungen

Wir können den Ausdruck klammern.

$$3 + (4 * 5)$$

# Einfache Auswertungen

Wir können den Ausdruck auch vollständig klammern:

$$(3 + (4 * 5))$$

Die Bedeutung ist noch dieselbe wie vom Anfang. Aber Abhängigkeiten sind jetzt explizit.

# Übrigens ...

Den Operator zwischen die Operanden zu schreiben ist eine willkürliche Wahl.

$$(1 + 2)$$

Der Operator kann auch vorangestellt werden.

$$(+ 1 2)$$

Oder nachgestellt

$$(1 2 +)$$

# Übrigens ...

Den Operator zwischen die Operanden zu schreiben ist eine willkürliche Wahl.

$$(1 + 2)$$

Der Operator kann auch vorangestellt werden.

$$(+ 1 2)$$

Oder nachgestellt

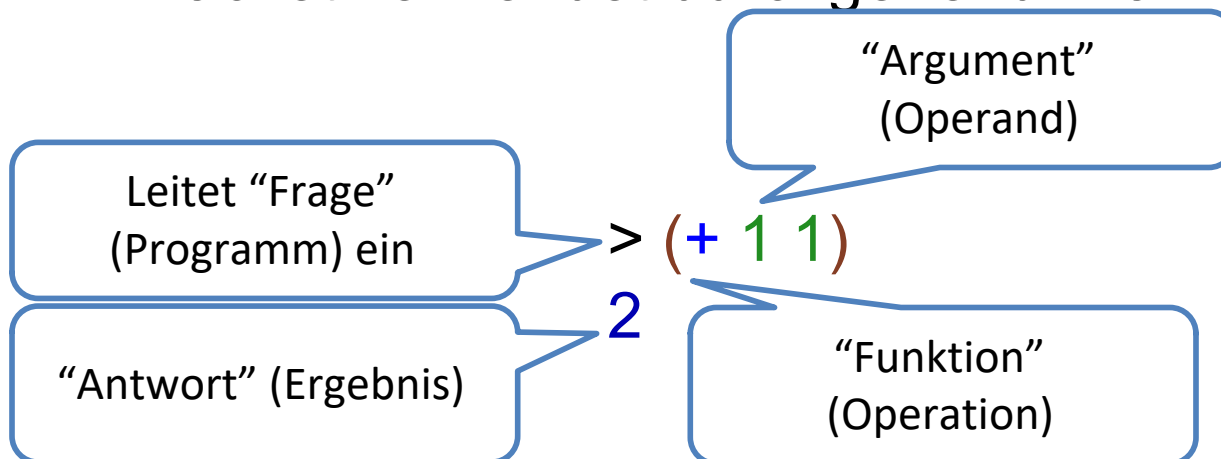
$$(1 2 +)$$

“Prefixnotation“ besonders gut computerlesbar.

Wir verwenden  
“Prefixnotation“ bereits bei  
manchen Operationen:  
-5  
sin(0)

# Arithmetik

- DrRacket Programme sind arithmetische Ausdrücke
- Das Ergebnis des Programms ist der ausgewertete Ausdruck
- DrRacket verwendet durchgehend Präfixnotation



# Beispiele

> (+ 2 2)

4

> (\* 3 3)

9

> (- 4 2)

2

> (/ 6 2)

3

> (sqr 3)

9

> (expt 2 3)

8

> (sin 0)

0

# Zahlen sind Ausdrücke

- Bislang: Argumente waren Zahlen
- Zahlen sind Ausdrücke, die zu sich selbst ausgewertet werden  
    > 5  
    5
- Zahlen sind "atomare" Ausdrücke



# Zusammengesetzte Ausdrücke

- Ausdrücke können als Argument verwendet werden

> (+ 2 (+ 3 4))  
9

# LiveVoting

PIN: IWMB

×

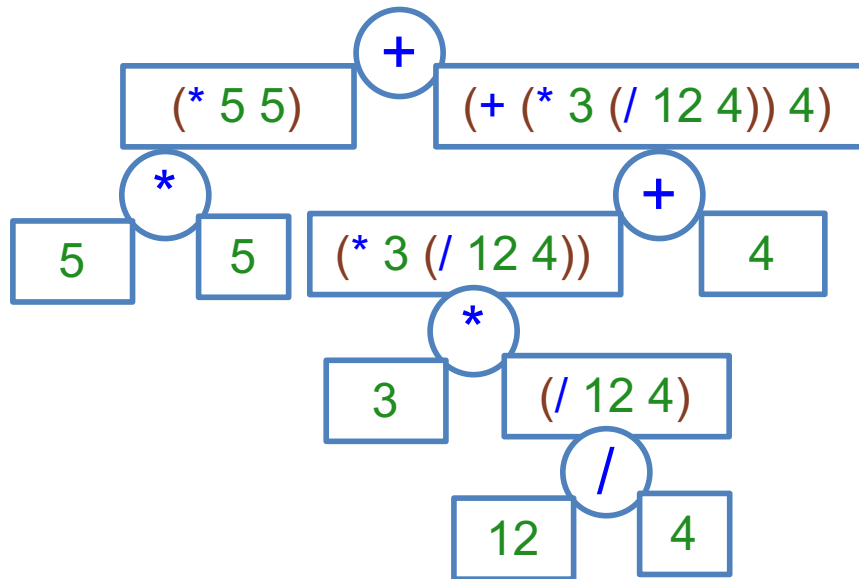
<https://ilias.uni-marburg.de/vote/IWMB>



# Auswertung von Ausdrücken

- Anwenden der Funktion auf die Werte der Argumente
- Argument ist Ausdruck: zunächst diesen Ausdruck auswerten (“rekursive Anwendung“ des Algorithmus)
- Auswertungsreihenfolge der Unterausdrücke ist irrelevant.

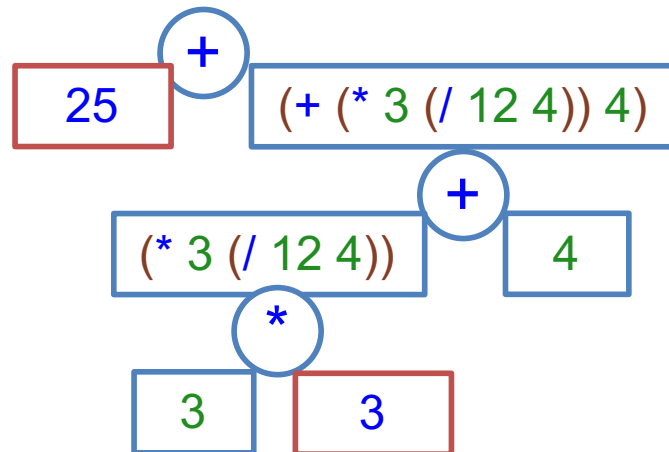
$(+ (* 5 5) (+ (* 3 (/ 12 4)) 4))$



# Auswertung von Ausdrücken

- Anwenden der Funktion auf die Werte der Argumente
- Argument ist Ausdruck: zunächst diesen Ausdruck auswerten (“rekursive Anwendung“ des Algorithmus)
- Auswertungsreihenfolge der Unterausdrücke ist irrelevant.

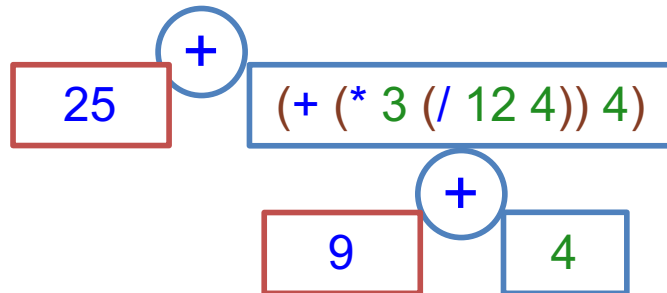
$(+ (* 5 5) (+ (* 3 (/ 12 4)) 4))$



# Auswertung von Ausdrücken

- Anwenden der Funktion auf die Werte der Argumente
- Argument ist Ausdruck: zunächst diesen Ausdruck auswerten (“rekursive Anwendung“ des Algorithmus)
- Auswertungsreihenfolge der Unterausdrücke ist irrelevant.

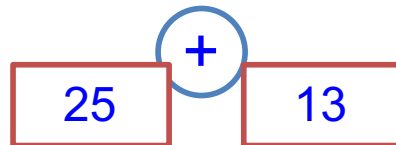
$(+ (* 5 5) (+ (* 3 (/ 12 4)) 4))$



# Auswertung von Ausdrücken

- Anwenden der Funktion auf die Werte der Argumente
- Argument ist Ausdruck: zunächst diesen Ausdruck auswerten (“rekursive Anwendung“ des Algorithmus)
- Auswertungsreihenfolge der Unterausdrücke ist irrelevant.

$(+ (* 5 5) (+ (* 3 (/ 12 4)) 4))$



# Auswertung von Ausdrücken

- Anwenden der Funktion auf die Werte der Argumente
- Argument ist Ausdruck: zunächst diesen Ausdruck auswerten (“rekursive Anwendung“ des Algorithmus)
- Auswertungsreihenfolge der Unterausdrücke ist irrelevant.

$(+ (* 5 5) (+ (* 3 (/ 12 4)) 4))$

38

# Literale

- Literale:
  - Werte, die direkt im Programmcode auftauchen
  - “Atomare Ausdrücke“
- Bisher: Zahlenliterale
- Andere Sorten von Literalen
  - Text
  - Wahrheitswerte
  - Bilder



# Datentypen

- Alle Werte lassen sich in "Datentypen" (Sorten) einteilen
- Auf Werte vom selben Datentyp lassen sich dieselben Funktionen anwenden (das Ergebnis ist evtl. unterschiedlich)
- Z.B. lassen sich Zahlen addieren, Texte nicht

# Datentypen - Konstruktoren

- Werte lassen sich z.B. durch Literale erzeugen
- 3
- 5.3
- “Deklarative Programmierung”
- true

# Datentypen - Funktionen

- Funktionen erwarten Argumente, die Werte von bestimmten Datentypen zurückliefern
- Eine "Funktion ist auf einem Datentyp definiert"
- Funktionen auf dem Datentyp String (Text):  
> (`string-append` "Deklarative " "Programmierung")  
"Deklarative Programmierung"
- Etc.
- → "Arithmetik der Strings"
- Liste der vordefinierten Funktionen:  
[https://docs.racket-lang.org/htdp-langs/beginner.html#\(part.\\_beginner-pre-defined\)](https://docs.racket-lang.org/htdp-langs/beginner.html#(part._beginner-pre-defined))

# Funktionen

- Bisher
  - Binäre Operationen
  - Datentyp von Argumenten und Ergebnis gleich
- Funktionen können beliebig viele Argumente haben

> (+ 2 3 4)

9

- Funktionen können Werte von unterschiedlichen Datentypen erwarten und liefern

> (+ (string-length "Programmiersprachen") 5)

24

19

# Funktionen

- Funktionen können Argumente unterschiedlicher Datentypen erwarten

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

- Funktionen können Datentypen umwandeln

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

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

# Repräsentationen

- Zahlenliteral

- 42

- Stringliteral

- "42"

- Ausdruck

- (+ 21 21)

- Stringliteral

- "(+ 21 21)"

Ähnlich, aber nicht kompatibel!

> (+ "42" 1)

+: expects a number as 1st argument, given "42"

# Datentyp: Wahrheitswerte

- Zwei Literale: `true`, `false`.
- Aussagenlogische Funktionen, z.B.
  - > `(and true true)`  
`true`
  - > `(and true false)`  
`false`
- Vergleiche, z.B.
  - > `(> 10 9)`  
`true`
  - > `(= 42 9)`  
`false`
  - > `(string=? "hello" "world")`  
`false`

# Datentyp: Bilder

- In DrRacket sind Bilder Werte
- Einfügen per Copy/Paste oder Einfügen → Bild einfügen
- Bilder sind Literale und werten zu sich selbst aus
- Weitere Funktionen für Bilder

# DEMO



# Datentyp: Bilder

- In DrRacket sind Bilder Werte
- Einfügen per Copy/Paste oder Einfügen → Bild einfügen
- Bilder sind Literale und werten zu sich selbst aus
- Weitere Funktionen für Bilder

```
> (* (image-width  ) (image-height  ))
```

```
600
```

```
> (circle 10 "solid" "red")
```



- Etc.

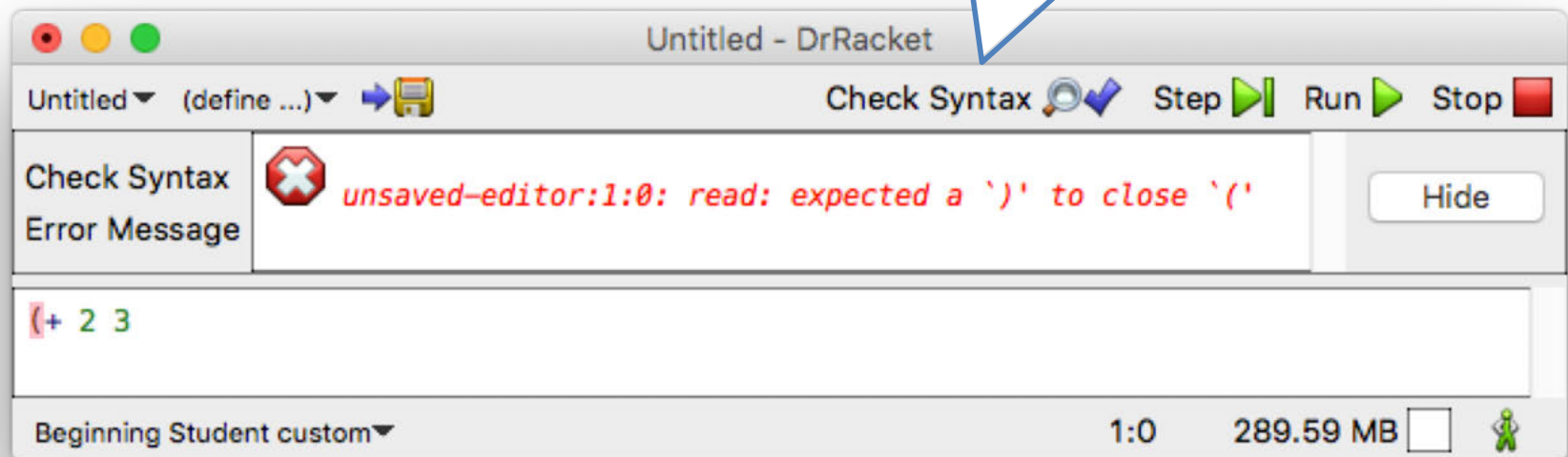
# Syntax

- Programmiersprachen haben eine Grammatik (Syntax)
- Die Grammatik gibt an, welche Struktur ein Programm haben muss
- Grammatik eines Dr Racket Beginning Student Language (BSL) Programms
  - Programm: Sequenz von Ausdrücken
  - Ausdruck: Literal oder Funktionsaufruf
  - Literal: Zahl, Bild, Wahrheitswert, String
  - Funktionsaufruf: (**f** **a1** **a2** ...), wobei f der Funktionsname und a1, a2, ... Ausdrücke sind

# Syntaxfehler

- Die Syntax kann überprüft werden, ohne das Programm auszuführen

Prüfen durch Klick auf  
"Check Syntax".



# Andere Fehler

- Nicht alle syntaktisch korrekten Programme haben eine Bedeutung
- D.H. nicht alle syntaktisch korrekten Ausdrücke lassen sich auswerten
- Fehler die kein Syntaxfehler sind, treten zur Laufzeit auf
  - > (number->string "asdf")  
number->string: expects a number, given "asdf"
  - > (string-length "asdf" "fdsa")  
string-length: expects only 1 argument, but found 2

Typfehler: tatsächlicher Typ und erwarteter Typ stimmen nicht überein.

# Andere Fehler

> (/ 1 0)

/: division by zero

Funktion ist nicht für alle Argumente definiert.

# Fehlercodes

- Fehler können auch unterdrückt werden
  - > (string->number "asdf")  
false
- "asdf" repräsentiert keine Zahl
  - Es tritt kein Laufzeitfehler auf
  - Stattdessen wird false zurückgeliefert
- Das Programm kann selbst entscheiden wie mit fehlerhaften Umwandlungen umgegangen wird

# Zeitpunkt von Fehlern

- Frühe Fehler sind einfacher zu beheben
  - Syntaxfehler < Laufzeitfehler < Fehlercodes
- Späte Fehler sind weniger restriktiv

# Bedeutung von Ausdrücken

- Ein BSL-Programm besteht aus Ausdrücken
- Beschreibung beliebiger BSL Programme durch Variablen (kursiv)
- Namenskonvention
  - Beginnend mit  $e$ : beliebiger Ausdruck ( $e, e_1, e', \dots$ )
  - Beginnend mit  $v$ : Wert, z.B. eine Zahl, ein Bild, etc. ( $v, v_1, v', \dots$ )



# Algorithmus zur Bestimmung der Bedeutung

1. Gegeben ein Ausdruck  $e$ , ist seine Bedeutung
  - a. Wenn  $e$  bereits ein Wert ist, so ist dieser Wert seine Bedeutung
  - b. Hat  $e$  die Form  $(f e_1 \dots e_n)$ , so wird  $e$  wie folgt ausgewertet
    - i. Sind  $e_1 \dots e_n$  bereits Werte  $v_1 \dots v_n$  und ist  $f$  auf  $v_1 \dots v_n$  definiert, dann ist der Wert von  $e$  die Anwendung von  $f$  auf  $v_1 \dots v_n$
    - ii. Sind  $e_1 \dots e_n$  sind bereits Werte  $v_1 \dots v_n$  aber ist  $f$  **nicht** auf  $v_1 \dots v_n$  definiert, dann wird die Auswertung mit einem Laufzeitfehler abgebrochen
    - iii. Ansonsten bestimme den Wert  $v_i$  für jeden noch nicht ausgewerteten Ausdruck  $e_i$  durch Anwenden von Schritt 1 und ersetze  $e_i$  durch  $v_i$ . Führe anschließend Schritt 1.b aus (jetzt ist entweder 1.b.i oder 1.b.ii anwendbar)

# Schrittweise Reduktion

- $e \rightarrow e'$ :  $e$  kann in einem Schritt zu  $e'$  reduziert werden
- Werte können nicht mehr reduziert werden
- Reduktion eines Ausdrucks  $e$ 
  1. Hat  $e$  die Form  $(f\ v_1 \dots v_n)$  und hat die Anwendung von  $f$  auf  $v_1 \dots v_n$  den Wert  $v$ ,  
dann gilt  $(f\ v_1 \dots v_n) \rightarrow v$ .
  2. Hat  $e$  einen Unterausdruck  $e_1$  in einer Auswertungsposition mit  $e_1 \rightarrow e_1'$ ,  
dann gilt  $e \rightarrow e'$ , wobei  $e'$  aus  $e$  erzeugt wird indem  $e_1$  durch  $e_1'$  ersetzt wird.
- Regel 2 wird "Kongruenzregel" genannt

# Schrittweise Reduktion

- $e \rightarrow e'$ :  $e$  kann in einem Schritt zu  $e'$  reduziert werden
- Werte können nicht mehr reduziert werden
- Reduktion eines Ausdrucks  $e$ 
  1. Hat  $e$  die Form  $(f v_1 \dots v_n)$  und hat die Anwendung von  $f$  auf  $v_1 \dots v_n$  den Wert  $v$ ,  
dann gilt  $(f v_1 \dots v_n) \rightarrow v$ .
  2. Hat  $e$  einen Unterausdruck  $e_1$  in einer Auswertungsposition mit  $e_1 \rightarrow e_1'$ ,  
dann gilt  $e \rightarrow e'$ , wenn  $e$  durch Ersetzen von  $e_1$  durch  $e_1'$  entsteht.
- Regel 2 wird "Kongruenz" genannt

Bislang sind alle Positionen auch Auswertungspositionen.

# Konventionen

- $e_1 \rightarrow e_2 \rightarrow e_3$  bedeutet:  $e_1 \rightarrow e_2$  und  $e_2 \rightarrow e_3$
- $e \rightarrow^* e'$  bedeutet:
  - Es gibt ein  $n \geq 0$  und  $e_1, \dots, e_n$ , sodass gilt  
 $e \rightarrow e_1 \rightarrow \dots \rightarrow e_n \rightarrow e'$
- Es gilt immer:  $e \rightarrow^* e$
- $\rightarrow^*$  wird der “reflexiv-transitive Abschluss von  $\rightarrow$ ” genannt  
(d.h. alle indirekt erreichbaren Reduktionen)

# Beispiele

- $(+ \ 1 \ 1) \rightarrow 2$
- Kongruenzregel kann in beliebiger Reihenfolge angewandt werden:
  - $(+ \ (* \ 2 \ 3) \ (* \ 4 \ 5)) \rightarrow (+ \ 6 \ (* \ 4 \ 5)) \rightarrow (+ \ 6 \ 20) \rightarrow 26$
  - $(+ \ (* \ 2 \ 3) \ (* \ 4 \ 5)) \rightarrow (+ \ (* \ 2 \ 3) \ 20) \rightarrow (+ \ 6 \ 20) \rightarrow 26$

# Konfluenz

- Anwendungsreihenfolge der Kongruenzregel beeinflusst Ergebnis der Auswertung nicht!
- Konfluenzregel:
  - Gegeben  $e_1 \rightarrow e_2$  und  $e_1 \rightarrow e_3$ ,  
dann gilt: es gibt ein  $e_4$ , sodass  $e_2 \rightarrow^* e_4$  und  $e_3 \rightarrow^* e_4$

# Äquivalenz

- Es gilt  $e_1 \equiv e_2$  ( $e_1$  ist äquivalent zu  $e_2$ ), falls ein Ausdruck  $e$  existiert, sodass  $e_1 \rightarrow^* e$  und  $e_2 \rightarrow^* e$
- Beispiele

$$(+ \ 1 \ 1) \equiv 2$$

$$(+ \ (* \ 2 \ 3) \ 20) \equiv (+ \ 6 \ (* \ 4 \ 5)) \equiv 26$$

- Die Bedeutung eines Programms verändert sich also nicht, wenn wir Unterausdrücke durch äquivalente Ausdrücke ersetzen

# LiveVoting

PIN: S9ZJ

✕

<https://ilias.uni-marburg.de/vote/S9ZJ>



# Funktionsdefinitionen

- Bislang: vordefinierte Funktionen
- Funktionen können auch definiert werden  
(benutzen Sie den Definitionsbereich von DrRacket)
- Warum Funktionen definieren?

```
> (/ (+ 12 17) 2)
```

```
14.5
```

```
> (/ (+ 100 200) 2)
```

```
150
```

# Funktionsdefinitionen

- Bislang: vordefinierte Funktionen
- Funktionen können auch definiert werden (benutzen Sie den Definitionsbereich von DrRacket)
- Warum Funktionen definieren?

> (/ (+ 12 17) 2)

14.5

Vermeidung von  
Redundanz

> (/ (+ 100 200) 2)

150

Bessere  
Verständlichkeit durch  
abstrakte Namen

# Funktionsdefinitionen

- Schlüsselwort “define” zur Unterscheidung von Ausdrücken

```
(define (average x y) (/ (+ x y) 2))
```

- (Selbst-)Definierte Funktionen können benutzt werden wie vordefinierte Funktionen

```
> (average 12 17)
```

```
14.5
```

```
> (average 100 200)
```

```
150
```

# Funktionsdefinitionen

(define (FunctionName InputName1 InputName2 ...) BodyExpression)

- define
  - Schlüsselwort leitet Funktionsdefinition ein
- FunctionName
  - Namen über den die Funktion aufgerufen werden kann
- InputName1, InputName2, ...
  - Namen für die übergebenen Argumente
- BodyExpression
  - Ausdruck, der bestimmt was die Anwendung der Funktion auf die Argumente bedeutet
  - Verwendet in der Regel InputName1, InputName2, ...
  - Auch "Implementation" oder "Body" der Funktion

# Funktionsdefinitionen

- Funktionsdefinitionen sind keine Ausdrücke
  - Können nicht ausgewertet werden
  - Können nicht als Argument übergeben werden

# Aufruf von Benutzer-definierten Funktionen

(FunctionName ArgumentExpression1 ArgumentExpression2 ...)

- FunctionName
  - Name einer definierten Funktion
- ArgumentExpression1, ArgumentExpression2, ...
  - Ausdrücke, auf die die Funktion angewendet werden soll
- Aufrufe von Benutzer-definierten und vordefinierten Funktionen unterscheiden sich nicht