

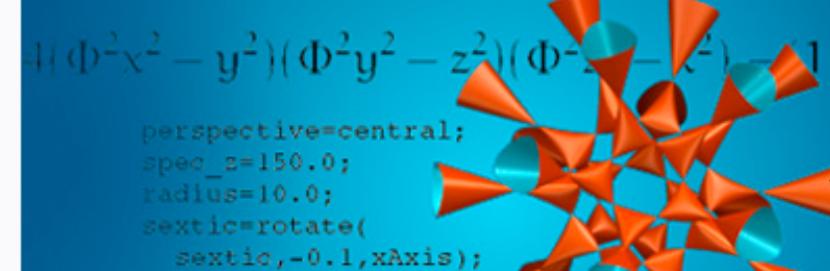
Deklarative Programmierung

Sommersemester 2018

Prof. Christoph Bockisch

(Programmiersprachen und –werkzeuge)

Steffen Dick, Alexander Bille, Johannes Frankenau,
Patrick Frömel, Niclas Schmidt, Jonas Stettin,
Robert Tran, Julian Velten



[Skript 8]

Bedeutung von BSL

- Bisher: informelle Definition der Bedeutung von BSL Programmen
 - Abstrakte Syntax: “Literale”, “Funktionsaufrufe”, etc.
 - Textuelle Beschreibung der Reduktionsschritte
- Bisher noch keine Definition der Bedeutung von struct
- Ab jetzt: **formale Definition** der Bedeutung von BSL Programmen
 - Konkrete Syntax
 - Formale Semantik basierend auf Syntax

Warum formale Definition?

- Eindeutige Festlegung
- Für den Computer
 - Ermöglicht erst Ausführung
 - Standardisierung
 - Analyse von Programmen
- Für den Programmierer
 - Korrekte Vorhersage des Ergebnisses eines Programm möglich
 - Systematisches Erlernen aller Programmiersprach-Konstrukte möglich

Syntax

- Abstrakte Syntax
 - Programm besteht aus Definitionen und Ausdrücken
 - Ausdruck besteht aus Literalen und Funktionsaufrufen
 - Etc.
- Konkrete Syntax
 - Wie genau schreibt man eine Zahl auf?
 - Wie können in einer Sequenz Definitionen von Ausdrücken unterschieden werden?
- Syntax-Definition durch Grammatik
 - Erkennen, ob ein Text der Syntax entspricht
 - Zerlegung eines gültigen Texts in Elemente der Sprache

Semantik

- Regeln für die Ausführung/Auswertung eines Programms
- Durch Reduktionsschritte
- Formale Definition: "Reduktionssemantik" (oder "strukturelle operationelle Semantik" oder "Plotkin Semantik")
- Treffen von logischen Aussagen, aus denen formale Beweise konstruiert werden können
- Aussagen basieren auf durch Grammatik vorgegebener Programmstruktur

Kontext-freie Grammatiken

- Verschiedene Klassen von Grammatiken
 - Ausdrucksmächtigkeit
 - Aufwand für die Zerlegung eines Satzes
- Für Programmiersprachen typischerweise idealer Trade-off: “Kontext-freie Grammatiken”
 - D.h. bisher eingelesener Text verändert die angewandte Grammatik nicht
 - Beim Anwenden der Grammatik-Regeln wird Zustand verwaltet: Dadurch Schachtelung möglich
- Prominente Notation für Kontext-freie Grammatiken: Erweiterte Backus Naur Form (EBNF)

Grammatik für Programmiersprachen

- Eingabe: Zeichenstrom

- Grammatik gibt an, welche Zeichen in welcher Reihenfolge erlaubt sind

Nichtterminal

Terminal

Alternative

- $\langle \text{ZifferNichtNull} \rangle ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'$

Eine “Produktion” pro Nichtterminal

- $\langle \text{ZifferNichtNull} \rangle$ akzeptiert einen Zeichenstrom, wenn er exakt aus einer der Ziffern 1-9 besteht

Grammatik für Programmiersprachen

- Verwendung von Nichtterminalen auf der rechten Seite der Produktion
- $\langle \text{Ziffer} \rangle ::= '0' \mid \langle \text{ZifferNichtNull} \rangle$

Jeder Zeichenstrom, der durch Produktion
 $\langle \text{ZifferNichtNull} \rangle$ akzeptiert wird.

Grammatik für Programmiersprachen

- Wiederholungen

- $\langle \text{GanzeZahl} \rangle ::= \langle \text{ZifferNichtNull} \rangle \langle \text{Ziffer} \rangle^*$

| '0'

*: Beliebig oft wiederholt,
kann auch ganz wegfallen.

- $\langle \text{Kommazahl} \rangle ::= \langle \text{GanzeZahl} \rangle \text{ '}' \langle \text{Ziffer} \rangle^+$

+: Beliebig oft wiederholt,
aber mindestens einmal.

Kontext-freie Grammatik für Zahlen

```
<Zahl>          ::= <PositiveZahl>
                  | '-' <PositiveZahl>
<PositiveZahl> ::= <GanzeZahl>
                  | <KommaZahl>
<GanzeZahl>    ::= <ZifferNichtNull> <Ziffer>*
                  | '0'
<Kommazahl>   ::= <GanzeZahl> '.' <Ziffer>+
<ZifferNichtNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<Ziffer>        ::= '0' | <ZifferNichtNull>
```

Welche Eingaben sind gültig?

- -87
- -.65
- 0
- -2.09900
- 007

Kontext-freie Grammatik für Zahlen

```
<Zahl>      ::= <PositiveZahl>
              | '-' <PositiveZahl>
<PositiveZahl> ::= <GanzeZahl>
                  | <KommaZahl>
<GanzeZahl>  ::= <ZifferNichtNull> <Ziffer>*
                  | '0'
<Kommazahl>  ::= <GanzeZahl> '.' <Ziffer>+
<ZifferNichtNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<Ziffer>       ::= '0' | <ZifferNichtNull>
```

Welche Eingaben sind gültig?

- -87
- ~~-05~~
- 0
- -2.09900
- ~~007~~

Vorzeichen gefolgt von positiver,
ganzer Zahl aus Ziffern 8 und 7.

Eine ganze Zahl muss mit 1-9 anfangen.

Ableitungsbaum

- Ableitung eines Nichtterminals
 - Auswahl einer Alternative der Produktion
 - Ersetzen aller Nichtterminale durch deren Ableitung

- Mögliche Ableitungen

von `<Ziffer>`

- '0'
- '3'

1. Alternative

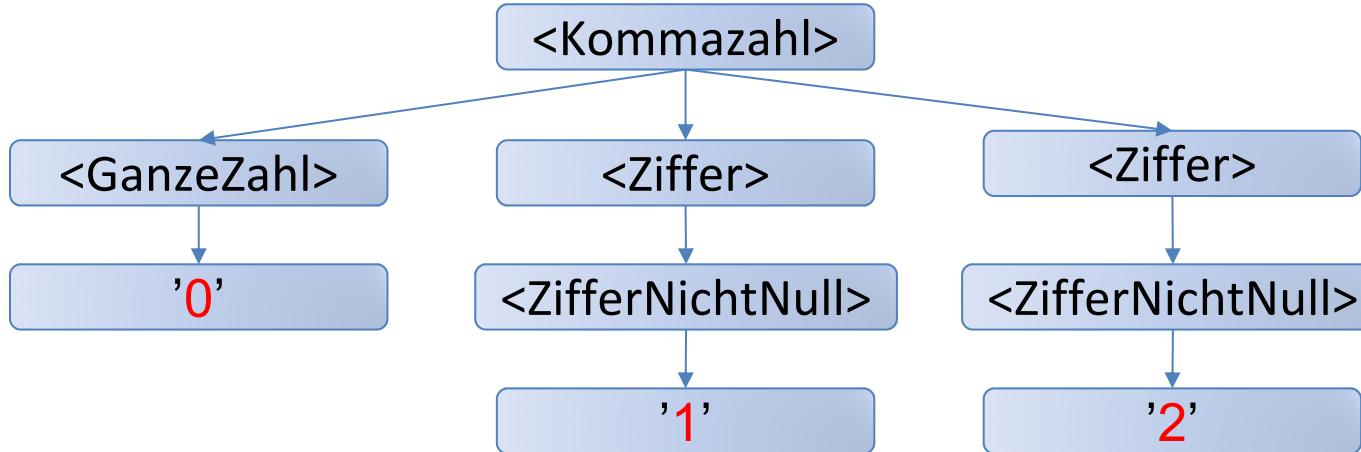
2. Alternative und Ableitung von
`<ZifferNichtNull>` zu 3. Alternative

```
<ZifferNichtNull> ::= '1' | '2' | '3' | '4' | '5'  
                      | '6' | '7' | '8' | '9'  
<Ziffer>           ::= '0' | <ZifferNichtNull>
```

Ableitungsbaum

- Mitprotokollieren, welche Nichtterminale abgeleitet wurden
→ Baum

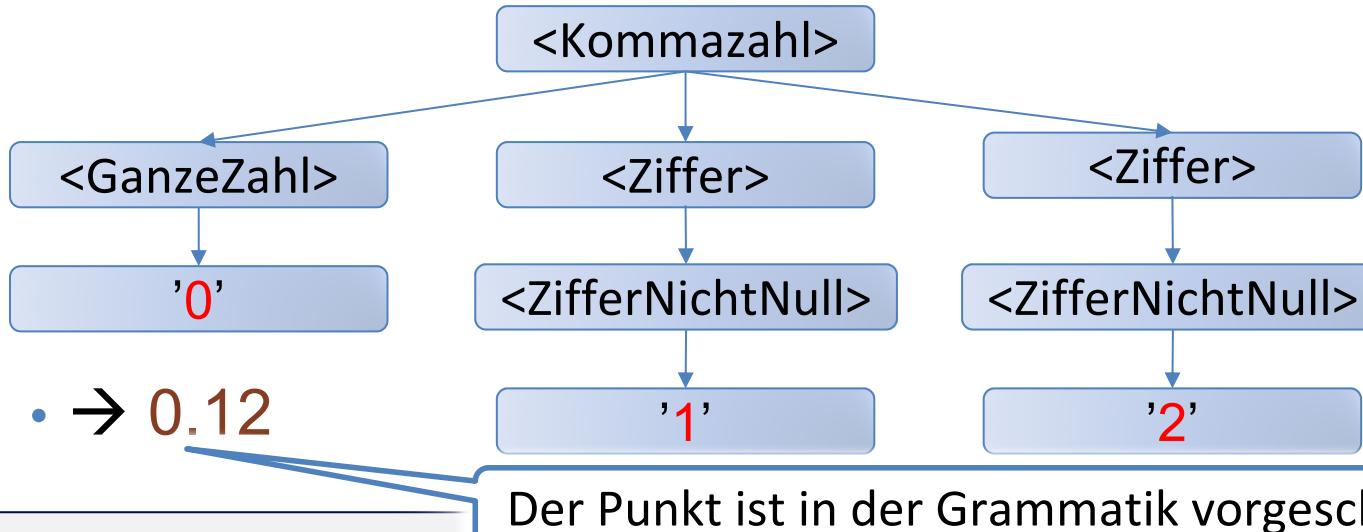
```
<GanzeZahl> ::= <ZifferNichtNull> <Ziffer>*  
                  | '0'  
<Kommazahl>  ::= <GanzeZahl> '.' <Ziffer>+  
<ZifferNichtNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'  
<Ziffer>       ::= '0' | <ZifferNichtNull>
```



Ableitungsbaum

- Ein Ableitungsbaum entspricht einem in der Grammatik gültigen Satz
- Im Allgemeinen: beliebig viele Ableitungsbäume
- Satz kann aus Ableitungsbaum abgelesen werden: Blätter von links nach rechts

```
<Kommazahl> ::= <GanzeZahl> '.' <Ziffer>+
```

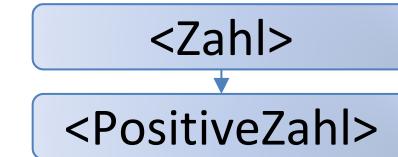


Ableitungsbäume

- Erzeugen aller gültiger Sätze
- Umgekehrt: prüfen, ob ein Satz gültig ist
 - Suchen einer Ableitung, die dem Satz entspricht
 - Ausgehend von Start-Produktion `<Zahl>`

- Beispiel 420

Kein '-': nur erste Alternative möglich



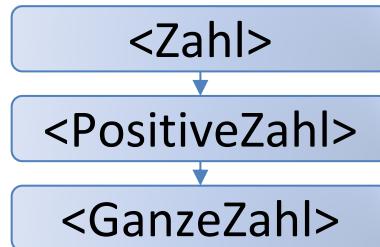
```
<Zahl> ::= <PositiveZahl>
          | '-' <PositiveZahl>
<PositiveZahl> ::= <GanzeZahl>
                   | <KommaZahl>
<GanzeZahl> ::= <ZifferNichtNull> <Ziffer>*
                  | '0'
<KommaZahl> ::= <GanzeZahl> '.' <Ziffer>+
<ZifferNichtNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<Ziffer> ::= '0' | <ZifferNichtNull>
```

Ableitungsbäume

- Erzeugen aller gültiger Sätze
- Umgekehrt: prüfen, ob ein Satz gültig ist
 - Suchen einer Ableitung, die dem Satz entspricht
 - Ausgehend von Start-Produktion `<Zahl>`

- Beispiel 420

Alternative noch
nicht eindeutig.



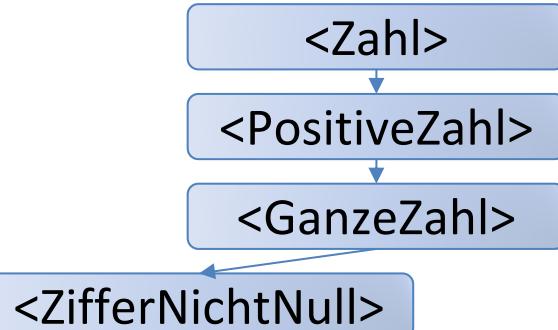
```
<Zahl> ::= <PositiveZahl>
          | '-' <PositiveZahl>
<PositiveZahl> ::= <GanzeZahl>
                   | <Kommazahl>
<GanzeZahl> ::= <ZifferNichtNull> <Ziffer>*
                  | '0'
<Kommazahl> ::= <GanzeZahl> '.' <Ziffer>+
<ZifferNichtNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<Ziffer> ::= '0' | <ZifferNichtNull>
```

Ableitungsbäume

- Erzeugen aller gültiger Sätze
- Umgekehrt: prüfen, ob ein Satz gültig ist
 - Suchen einer Ableitung, die dem Satz entspricht
 - Ausgehend von Start-Produktion `<Zahl>`

- Beispiel **420**

Kein '0': nur erste Alternative möglich



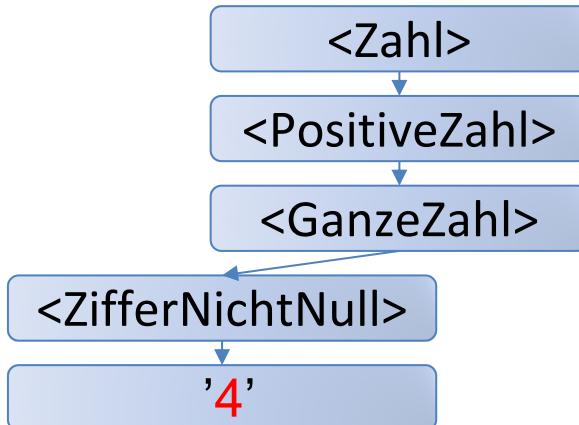
<code><Zahl></code>	<code>::= <PositiveZahl></code>
	<code> '-' <PositiveZahl></code>
<code><PositiveZahl></code>	<code>::= <GanzeZahl></code>
	<code> <Kommazahl></code>
<code><GanzeZahl></code>	<code>::= <ZifferNichtNull> <Ziffer>*</code>
	<code> '0'</code>
<code><Kommazahl></code>	<code>::= <GanzeZahl> '.' <Ziffer>+</code>
<code><ZifferNichtNull></code>	<code>::= '1' '2' '3' '4' '5' '6' '7' '8' '9'</code>
<code><Ziffer></code>	<code>::= '0' <ZifferNichtNull></code>

Ableitungsbäume

- Erzeugen aller gültiger Sätze
- Umgekehrt: prüfen, ob ein Satz gültig ist
 - Suchen einer Ableitung, die dem Satz entspricht
 - Ausgehend von Start-Produktion `<Zahl>`

- Beispiel 420

‘4’: nur vierte Alternative möglich



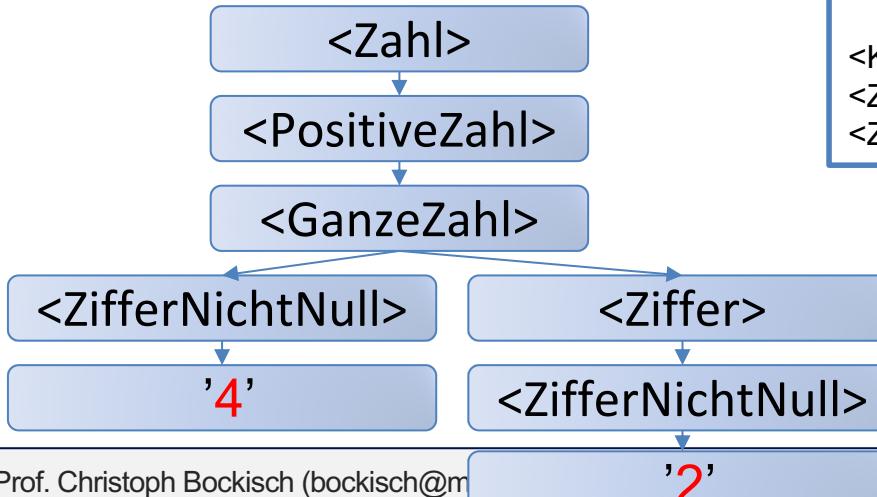
<code><Zahl></code>	<code>::= <PositiveZahl></code>
	<code> ‘-’ <PositiveZahl></code>
<code><PositiveZahl></code>	<code>::= <GanzeZahl></code>
	<code> <KommaZahl></code>
<code><GanzeZahl></code>	<code>::= <ZifferNichtNull> <Ziffer>*</code>
	<code> '0'</code>
<code><KommaZahl></code>	<code>::= <GanzeZahl> ‘.’ <Ziffer>+</code>
<code><ZifferNichtNull></code>	<code>::= '1' '2' '3' '4' '5' '6' '7' '8' '9'</code>
<code><Ziffer></code>	<code>::= '0' <ZifferNichtNull></code>

Ableitungsbäume

- Erzeugen aller gültiger Sätze
- Umgekehrt: prüfen, ob ein Satz gültig ist
 - Suchen einer Ableitung, die dem Satz entspricht
 - Ausgehend von Start-Produktion `<Zahl>`

- Beispiel 420

Eingabe noch nicht zu Ende.



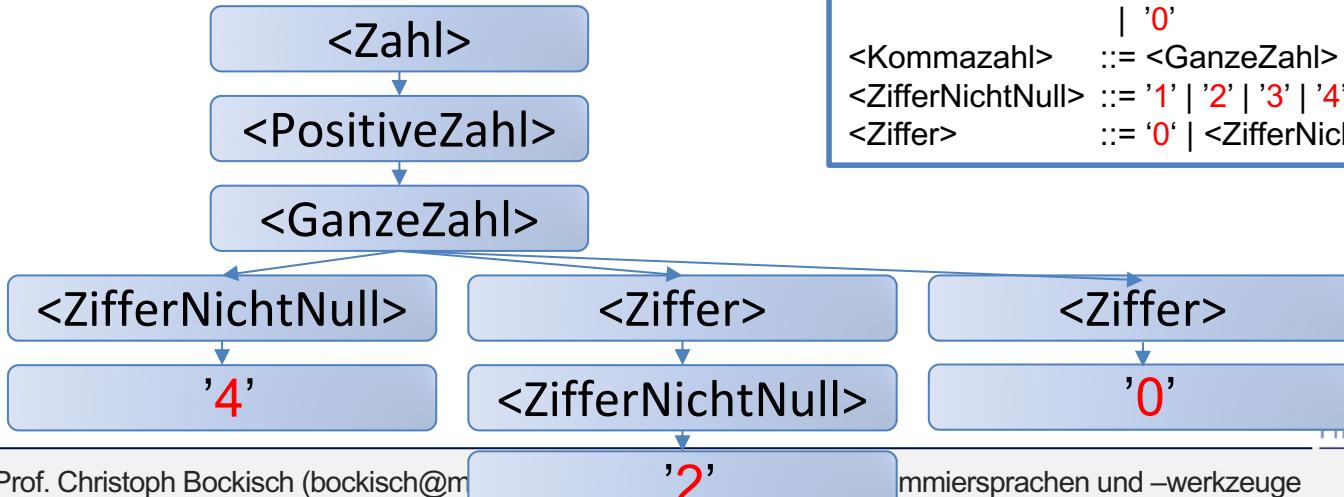
<code><Zahl></code>	<code>::= <PositiveZahl></code>
	<code> '-' <PositiveZahl></code>
<code><PositiveZahl></code>	<code>::= <GanzeZahl></code>
	<code> <Kommazahl></code>
<code><GanzeZahl></code>	<code>::= <ZifferNichtNull> <Ziffer>*</code>
	<code> '0'</code>
<code><Kommazahl></code>	<code>::= <GanzeZahl> '.' <Ziffer>+</code>
<code><ZifferNichtNull></code>	<code>::= '1' '2' '3' '4' '5' '6' '7' '8' '9'</code>
<code><Ziffer></code>	<code>::= '0' <ZifferNichtNull></code>

Ableitungsbäume

- Erzeugen aller gültiger Sätze
- Umgekehrt: prüfen, ob ein Satz gültig ist
 - Suchen einer Ableitung, die dem Satz entspricht
 - Ausgehend von Start-Produktion `<Zahl>`

- Beispiel 420

Eingabe zu Ende.



Ableitungsbäume

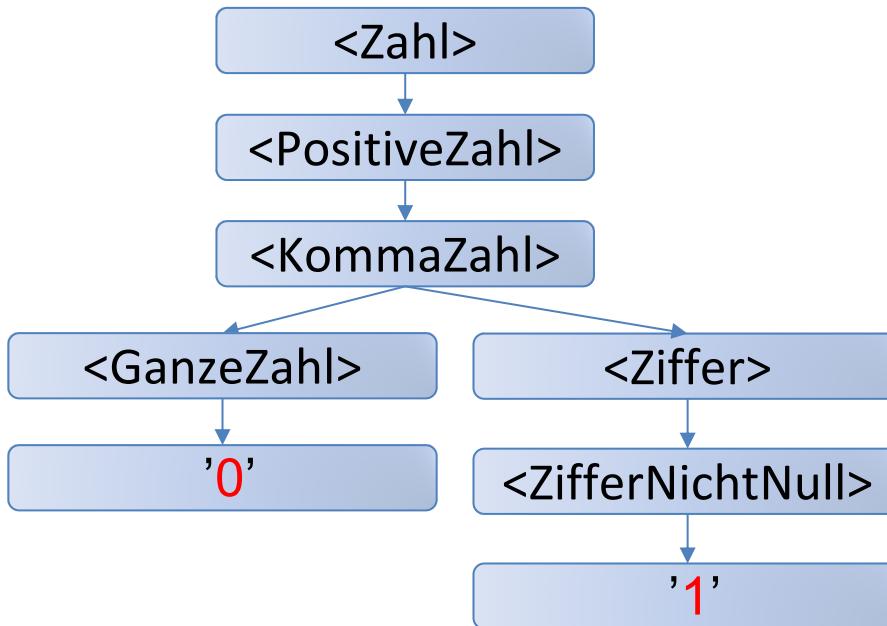
- Backtracking beim Erzeugen des Ableitungsbaums
- Beispiel 0.1



```
<Zahl>      ::= <PositiveZahl>
                  | '-' <PositiveZahl>
<PositiveZahl> ::= <GanzeZahl>
                  | <Kommazahl>
<GanzeZahl>  ::= <ZifferNichtNull> <Ziffer>*
                  | '0'
<Kommazahl>  ::= <GanzeZahl> '.' <Ziffer>+
<ZifferNichtNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<Ziffer>       ::= '0' | <ZifferNichtNull>
```

Ableitungsbäume

- Backtracking beim Erzeugen des Ableitungsbäums
- Beispiel 0.1

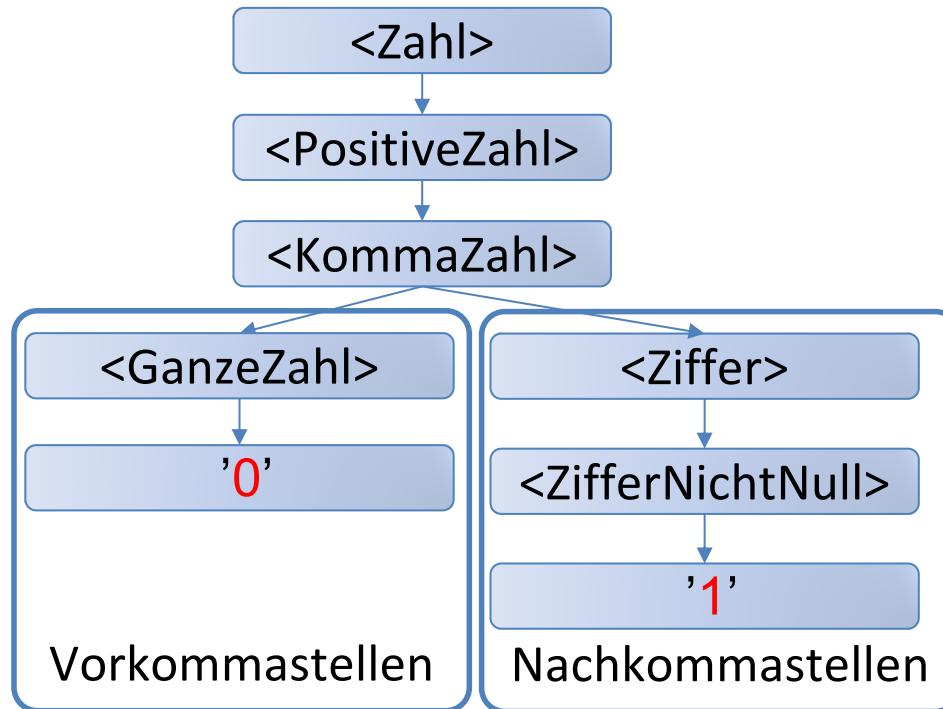


```

<Zahl> ::= <PositiveZahl>
          | '-' <PositiveZahl>
<PositiveZahl> ::= <GanzeZahl>
                   | <KommaZahl>
<GanzeZahl> ::= <ZifferNichtNull> <Ziffer>*
                  | '0'
<Kommazahl> ::= <GanzeZahl> '.' <Ziffer>+
<ZifferNichtNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<Ziffer> ::= '0' | <ZifferNichtNull>
  
```

Ableitungsbäume und Programmstruktur

- Programmstruktur in Ableitungsbäum erkennbar
- Beispiel 0.1



```

<Zahl>      ::= <PositiveZahl>
              | '-' <PositiveZahl>
<PositiveZahl> ::= <GanzeZahl>
                  | <KommaZahl>
<GanzeZahl>  ::= <ZifferNichtNull> <Ziffer>*
                  | '0'
<Kommazahl>  ::= <GanzeZahl> '.' <Ziffer>+
<ZifferNichtNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<Ziffer>       ::= '0' | <ZifferNichtNull>
  
```

Arten von Syntax

- Abstrakte Syntax
 - Struktur
 - Unabhängig von Codierung
 - Trenner von Tokens werden ignoriert (Whitespace, Kommentare)
 - Schlüsselwörter werden nicht spezifiziert
- Konkrete Syntax
 - Codierung
 - Whitespace, Kommentare
 - Bezeichner für Schlüsselwörter

Andere Verwendung des
Begriffs im Skript.

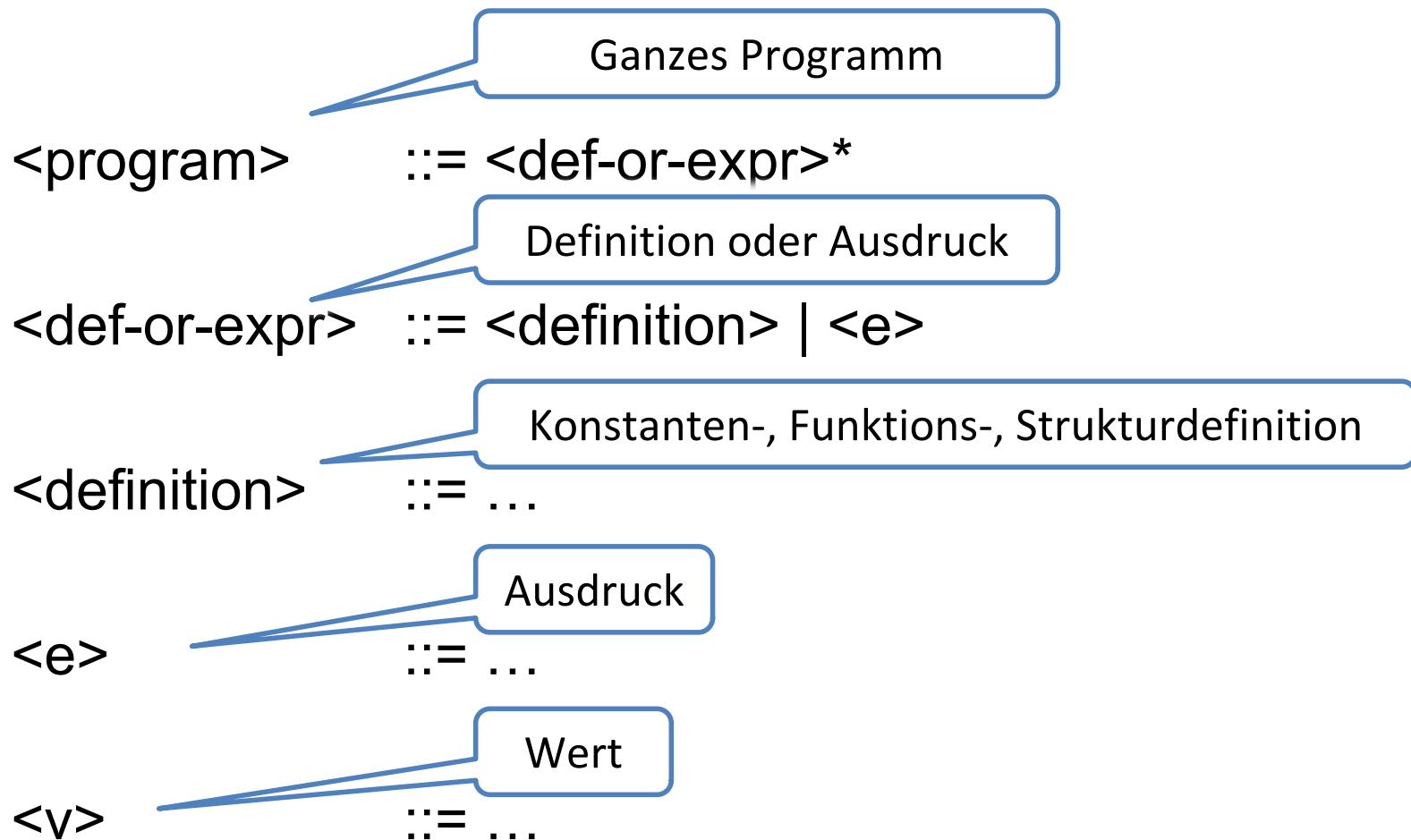
Unterscheidung von konkreter und
abstrakter Syntax nicht Klausur-relevant.



Abstrakter Syntaxbaum

- Ableitungsbaum bei abstrakter Grammatik
- → Abstrakter Syntaxbaum (AST)
- Reduktionsregeln:
 - Schrittweise Umschreiben von AST

Syntax von BSL



Syntax von BSL

```
<definition> ::= '(' 'define' '(' <name> <name>+ ')' <e> ')'  
          | '(' 'define' <name> <e> ')'  
          | '(' 'define-struct' <name> '(' <name>+ ')' ')'
```

Syntax von BSL

```
<e> ::= '(' <name> <e>+ ')'  
|  (' 'cond' {[<e> <e> ]}+ ')'  
|  (' 'cond' {[<e> <e> ]}* '[' 'else' <e> '] ')'  
|  (' 'if' <e> <e> <e> ')'  
|  (' 'and' <e> <e>+ ')'  
|  (' 'or' <e> <e>+ ')'  
|  <name>  
|  <v>
```

Geschweifte Klammern
um Wiederholungen von
Sequenzen anzugeben.

Syntax von BSL

```
<e> ::= '(' <name> <e>+ ')'  
|  (' 'cond' {[<e> <e> ]}+ ')'  
|  (' 'cond' {[<e> <e> ]}* '[' 'else' <e> '] ')'  
|  (' 'if' <e> <e> <e> ')'  
|  (' 'and' <e> <e>+ ')'  
|  (' 'or' <e> <e>+ ')'  
|  <name>  
|  <v>
```

Warum wird and/or von
Funktionsaufruf unterschieden?

Auswertung der Argumente
anders: Auswertung nur bis
Resultat feststeht.

Syntax von BSL

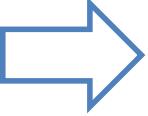
```
<v> ::= '<' 'make-'<name> <v>* '>'  
| <number>  
| <boolean>  
| <string>  
| <image>
```

Darf nicht tatsächlich in BSL
Programm vorkommen.
Verwendung bei AST-
Umschreiben durch
Reduktionsregeln.
Nicht zu verwechseln mit
Funktionsaufruf zur Erzeugung
einer Struktur-Instanz.

Syntax von BSL

- Nichtterminale ohne angegebene Produktion
- <name>
 - gültige Bezeichner für Funktionen, Konstanten, Strukturen und Parameter
- <number>
 - Zahlen, siehe vorige Vorlesung
- <boolean>
 - Wahrheitswerte **true** oder **false**
- <string>
 - Zeichenkette in Anführungszeichen
- <image>
 - Bildliterale

BSL Kernsprache

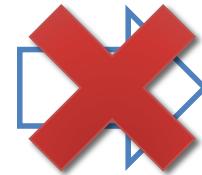
- Vereinfachung der Syntax durch Weglassen von syntaktischem Zucker
- Bereits bekannt:
 - if
 - else
- Zusätzlich
 - $(\text{or } e_1 \dots e_n)$  $(\text{not } (\text{and } (\text{not } e_1) \dots (\text{not } e_n)))$
- BSL Kernsprache ist Teilmenge der BSL Sprache
 - BSL Programme können durch Transformation auf gleichwertige BSL Kernsprache Programme abgebildet werden

Ist and syntaktischer Zucker?

- Auswertung von and
- Auswertung der Teilausdrücke bis
 - Ein Ausdruck false ergibt → Gesamtausdruck ist false
 - Alle Ausdrücke zu true ausgewertet sind → Gesamtausdruck is true
- Einschränkung der Auswertungspositionen bekannt von cond Ausdrücken
- Kann and durch cond realisiert werden?
 - → Nein!

>(and true 42)

*and: question result
is not true or false: 42*



>(cond [true 42] [else false])

42

Syntax von Kern BSL

- Nur Grammatik für Ausdrücke ist verändert

```
<e>    ::= '(' <name> <e>+ ')'  
        | '(' 'cond' {[ '<e> <e>' ]}+ ')' (red)  
        | '(' 'and' <e> <e>+ ')' (red)  
        | <name>  
        | <v>
```

Mögliche Programmergebnisse

- Bedeutung eines Programms: Sequenz von Reduktionsschritten
- Mögliche Resultate
 - Reduktion zu einem Wert
 - Abbruch mit Fehler
 - Nicht terminierende Sequenz
- Werte:
 - Alle Konstanten (Literale)
 - Instanzen von Strukturen

Instanzen von Strukturen

- Darstellung: ‘<‘ ‘make-‘<name> <v>* ’>‘
- Unterscheidung
 - Ausdruck zur Erzeugung mit runden Klammern
`(make-posn 3 (+ 2 2))`
 - Wert einer Instanz mit spitzen Klammern
`<make-posn 3 4>`

Umgebung

- Eigenschaft eines Ausdrucks im Programm
- Enthält bekannte Definitionen
- Abhängig von der Position des Ausdrucks
- Umgebung eines Ausdrucks enthält alle Definitionen die vor dem aktuell ausgewerteten Ausdruck stehen

Darstellung der Umgebung

- Struktur der Umgebung kann ebenfalls durch Grammatik definiert werden

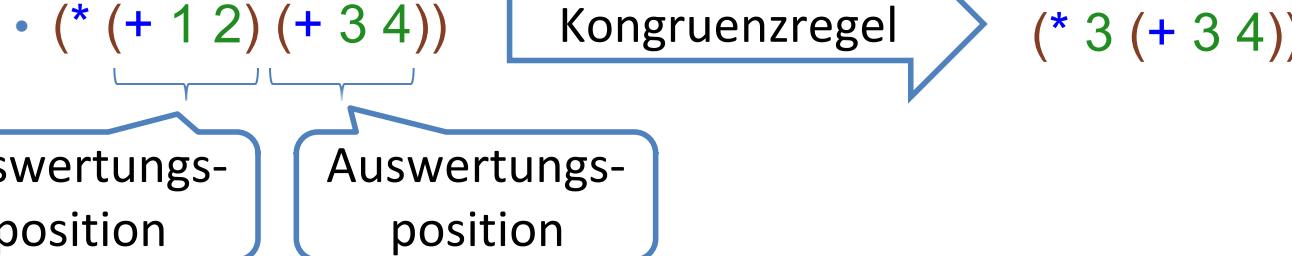
`<env> ::= <env-element>*`

`<env-element> ::= (' ' define ' (' <name> <name>+ ')' <e> ')'`
| ' ' define <name> <v> ')'
| ' ' define-struct <name> ' (' <name>+ ')')'

Achtung: Konstanten sind in der Umgebung durch einen **Wert** definiert.
Im Gegensatz dazu sind sie in der BSL Sprache durch einen **Ausdruck** definiert.

Wiederholung

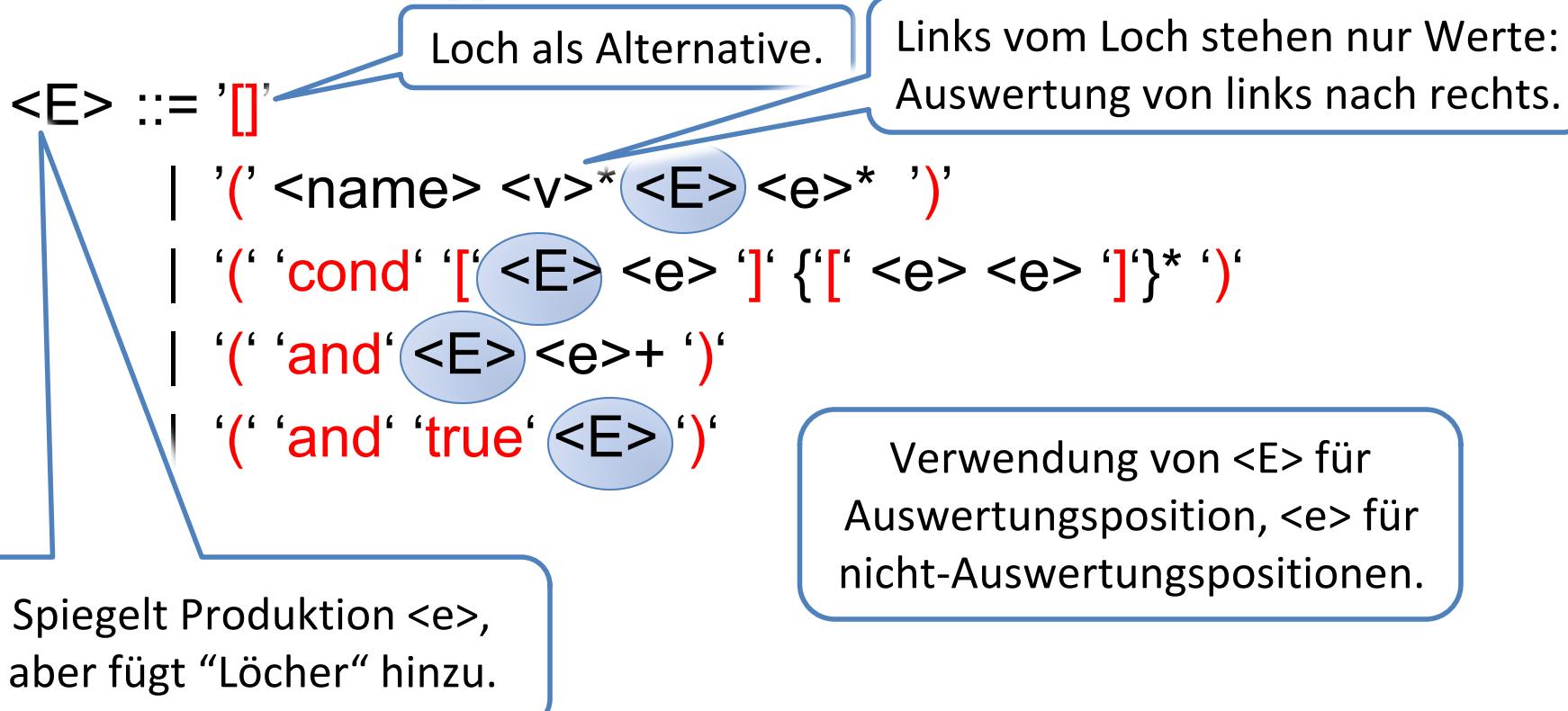
- Auswertungspositionen
 - Markiert Teilausdrücke, die als nächstes ausgewertet werden dürfen
- Kongruenzregel
 - Hat e einen Unterausdruck e_2 in einer Auswertungsposition mit $e_2 \rightarrow e'_2$, dann gilt $e \rightarrow e'$, wobei e' aus e erzeugt wird indem e_2 durch e'_2 ersetzt wird.
 - Also: hat ein Ausdruck Teilausdrücke, so können diese ausgewertet werden und die Teilausdrücke durch deren Werte ersetzt werden.



Auswertungskontext

- Formalisierung von Kongruenzregel und Auswertungsposition
- Grammatik mit “Loch”: ‘[]’
- Jedes Element enthält genau ein Loch

Auswertungskontext



Auswertungskontext

- Was hiervon sind Auswertungskontexte?
- $(\text{posn}-x \ (\text{make-posn} \ 14 \ []))$
- $(\text{and} \ \text{true} \ (\text{and} \ [] \ x))$
- $(\text{make-posn} \ 14 \ 17)$
- $(\text{and} \ x \ [])$

Kein Loch

Loch an falscher
Position

Auswertungsreihenfolge

- Bisher: Auswertungspositionen können in beliebiger Reihenfolge ausgewertet werden
- Jetzt: Auswertungsreihenfolge von links nach rechts

Auswertungspositionen

- In einen Auswertungskontext können Ausdrücke eingesetzt werden
- $E[e]$
 - Ersetzen von Loch in Auswertungskontext durch Ausdruck e
- Beispiel
 - $E = (* [] (+ 3 4))$
 - $E[(+ 1 2)] = (* (+ 1 2) (+ 3 4)).$

Kongruenzregel mit Auswertungskontext

- **(KONG):** Falls $e_1 \rightarrow e_2$, dann $E[e_1] \rightarrow E[e_2]$.

- Beispiel

- $e = (* (+ 1 2) (+ 3 4))$
- $E = (* [] (+ 3 4))$
- $e_1 = (+ 1 2)$ mit $e = E[e_1]$
- Reduktion von e_1 : $(+ 1 2) \rightarrow 3$
- Deshalb Reduktion von e : $E[3] = (* 3 (+ 3 4))$

Namenskonventionen

- Bezeichner in Regeldefinitionen
 - Nichtterminal $\langle x \rangle$
 - Dann stehen Varianten von x für beliebige Worte des Nichtterminals
 (x_1, x_2, x, x')
- Diese Bezeichner sind “Metavariablen”
 - Meta: Variable nicht über Werte im Programm, sondern über Programmteile

Mengenschreibweise

- Nichtterminal als Menge:
Menge aller Worte, die aus Nichtterminal ableitbar sind
- Mengenschreibweise für Kongruenzregel
Für alle $e_1 \in \langle e \rangle$ und alle $e_2 \in \langle e \rangle$ und alle $E \in \langle E \rangle$
Falls $e_1 \rightarrow e_2$, dann $E[e_1] \rightarrow E[e_2]$.

Bedeutung von Programmen

- **(PROG):** Ein Programm wird von links nach rechts ausgeführt und startet mit der leeren Umgebung. Ist das nächste Programmelement ...
 - ... eine Funktions- oder Strukturdefinition, so wird diese Definition in die Umgebung aufgenommen und die Ausführung mit dem nächsten Programmelement in der erweiterten Umgebung fortgesetzt.
 - ... ein Ausdruck, so wird dieser gemäß der nachfolgenden Regeln in der aktuellen Umgebung zu einem Wert ausgewertet.
 - ... eine Konstantendefinition (`define x e`), so wird in der aktuellen Umgebung zunächst `e` zu einem Wert `v` ausgewertet und dann `(define x v)` zur aktuellen Umgebung hinzugefügt.

Beispiel

```
(define (f x) (+ x 1))  
(define c (f 5))  
(+ c 3)
```

Umgebung

Beispiel



```
(define (f x) (+ x 1))  
(define c (f 5))  
(+ c 3)
```

Umgebung

```
(define (f x) (+ x 1))
```

Beispiel

```
(define (f x) (+ x 1))
```

```
→ (define c (f 5))
```

```
      (+ c 3)
```

In Umgebung
“(define (f x) (+ x 1))”
auswerten zu 6.

Umgebung

```
(define (f x) (+ x 1))  
(define c 6)
```

Beispiel

```
(define (f x) (+ x 1))
```

```
(define c (f 5))
```

```
(+ c 3)
```

In aktueller Umgebung
auswerten zu 9.

Umgebung

```
(define (f x) (+ x 1))  
(define c 6)
```

Auswertungsregeln

- Notation: Reduktionsregeln
- Auswertung eines Ausdrucks in Umgebung env
 - Env bei Reduktionsregel implizit gegeben
- Ausdruck e wird ausgewertet, indem er bis zu einem Wert reduziert wird:
 $e \rightarrow e_1 \rightarrow \dots \rightarrow v$
- Ausdruck kein Wert, aber keine Reduktionsregel anwendbar: Programmfehler

Reduktionsregeln für Funktionsaufruf

- Primitive Funktionen
 - Nicht in Umgebung definiert
 - Auswertung eingebaut
 - **(PRIM):** Falls name eine primitive Funktion f ist und $f(v_1, \dots, v_n) = v$, dann $(\text{name } v_1 \dots v_n) \rightarrow v$
- Definierte Funktion
 - Definition in Umgebung enthalten
 - In Form eines Ausdrucks
 - **(FUN):** Falls $(\text{define} (\text{name } name_1, \dots, name_n) e)$ in Umgebung, dann $(\text{name } v_1 \dots v_n) \rightarrow e[name_1 := v_1 \dots name_n := v_n]$

Ersetze Vorkommen von formalen Parametern
in e durch die übergebenen Argumentwerte.

Reduktionsregel für Konstanten

- **(CONST):** Falls (define *name v*) in der Umgebung, dann
name → *v*

Reduktionsregeln für konditionale Ausdrücke

- Gemäß Auswertungskontext immer den ersten Bedingungsausdruck auswerten
- **(COND-True):** $(\text{cond} [\text{true } e] \dots) \rightarrow e$
- **(COND-False):** $(\text{cond} [\text{false } e_1] [e_2 e_3] \dots) \rightarrow (\text{cond} [e_2 e_3] \dots)$

Reduktionsregeln für boolesche Ausdrücke

- Wenn einer der Operanden **false** ist, ist der gesamte Ausdruck **false**
 - Auswertung bis zum ersten **false**

- **(AND-1):** $(\text{and} \text{ true } \text{true}) \rightarrow \text{true}$
- **(AND-2):** $(\text{and} \text{ true } \text{false}) \rightarrow \text{false}$
- **(AND-3):** $(\text{and} \text{ false } \dots) \rightarrow \text{false}$
- **(AND-4):** $(\text{and} \text{ true } e_1 \text{ } e_2 \dots) \rightarrow (\text{and } e_1 \text{ } e_2 \dots)$

Stellt sicher, dass alle Werte auch Wahrheitswerte sind.

Reduktionsregeln für Strukturen

- Strukturdefinition erzeugt
 - Konstruktor (*make-name*)
 - Selektoren (*name-field*)
 - Prädikate (*name?*)

Reduktionsregel für Konstruktoraufälle

- Ein Konstruktor kann aufgerufen werden, wenn ein entsprechendes Struct in der Umgebung definiert ist.
- Anzahl der Felder muss mit Anzahl der Argumente übereinstimmen
- (**STRUCT-make**):
Falls (define-struct **name** (**name**₁ ... **name**_n)) in der Umgebung, dann
(make-**name** **v**₁ ... **v**_n) → < make-**name** **v**₁ ... **v**_n >.

Reduktionsregel für Selektoraufälle

- Strukturdefinition bildet Feld-Namen auf Position innerhalb Struktur ab.

- `(define-struct vel (deltax deltay))`
- `(make-vel 1 2) → <make-vel 1 2>`

deltax steht an Position 1,
deltay steht an Position 2

- **(STRUCT-select):**
Falls `(define-struct name (name1 ... namen))` in der Umgebung, dann
$$(\text{name-name}_i < \text{make-name } v_1 \dots v_n >) \rightarrow v_i$$

Reduktionsregeln für Strukturprädikate

- (**STRUCT-predtrue**):

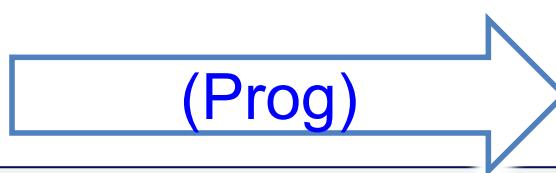
(**name?** < make-**name** ... >) → true

- (**STRUCT-predfalse**):

Falls **v** nicht < make-**name** ... >, dann (**name?** **v**) → false

Reduktion am Beispiel

```
(define-struct s (x y))  
(define (f x) (cond [(< x 1) (/ x 0)]  
                      [true (+ x 1)])  
                      [true x]))  
(define c (make-s 5 (+ (* 2 3) 4)))  
(f (s-x c))
```



(Prog)

Reduktion am Beispiel

```
(define-struct s (x y))  
(define (f x) (cond [(< x 1) (/ x 0)]  
                      [true (+ x 1)]  
                      [true x]))  
(define c (make-s 5 (+ (* 2 3) 4)))  
(f (s-x c))
```

env=

(Prog)

Reduktion am Beispiel

```
(define (f x) (cond [(< x 1) (/ x 0)]  
                      [true (+ x 1)]  
                      [true x]))  
  
(define c (make-s 5 (+ (* 2 3) 4)))  
  
(f (s-x c))
```

env=

```
(define-struct s (x y))
```

(Prog)

Reduktion am Beispiel

```
(define c (make-s 5 (+ (* 2 3) 4)))  
(f (s-x c))
```

env=

```
(define-struct s (x y))  
(define (f x) (cond [(< x 1) (/ x 0)]  
[true (+ x 1)])  
[true x]))
```

(Prog)

Reduktion am Beispiel

```
(define c (make-s 5 (+ (* 2 3) 4)))  
(f (s-x c))
```

e = (make-s 5 (+ (* 2 3) 4))
E = (make-s 5 (+ [] 4)) und e₁ = (* 2 3)
(PRIM) e₁ → 6
(KONG) e → (make-s 5 (+ 6 4))

env=

```
(define-struct s (x y))  
(define (f x) (cond [(< x 1) (/ x 0)]  
[true (+ x 1)])  
[true x]))
```

(Prog)

Reduktion am Beispiel

```
(define c (make-s 5 (+ 6 4)))  
(f (s-x c))
```

e = (make-s 5 (+ 6 4))
E = (make-s 5 []) und e₁ = (+ 6 4)
(PRIM) e₁ → 10
(KONG) e → (make-s 5 10)

env=

```
(define-struct s (x y))  
(define (f x) (cond [(< x 1) (/ x 0)]  
[true (+ x 1)])  
[true x]))
```

(Prog)

Reduktion am Beispiel

```
(define c (make-s 5 10))  
(f (s-x c))
```

```
(STRUCT-make)  
(make-s 5 10) → <make-s 5 10>
```

env=

```
(define-struct s (x y))  
(define (f x) (cond [(< x 1) (/ x 0)]  
[true (+ x 1)])  
[true x]))
```

(Prog)

Reduktion am Beispiel

(f (s-x c))

```
env=
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                     [true (+ x 1)])
                     [true x]))
(define c <make-s 5 10>)
```

(Prog)

Reduktion am Beispiel

(f (s-x c))

e = (f (s-x c))

E = (f (s-x [])) und e₁ = c

(CONST) c → <make-s 5 10>

(KONG) e → (f (s-x <make-s 5 10>))

env=

(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
[true (+ x 1)])
[true x]))
(define c <make-s 5 10>)

(Prog)

Reduktion am Beispiel

(f (s-x c))

e = (f (s-x c))

E = (f (s-x [])) und e₁ = c

(CONST) c → <make-s 5 10>

(KONG) e → (f (s-x <make-s 5 10>))

env=

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                     [true (+ x 1)])
                     [true x]))
(define c <make-s 5 10>)
```

(Prog)

Reduktion am Beispiel

(f (s-x <make-s 5 10>))

e = (f (s-x <make-s 5 10>))

E = (f []) und e₁ = (s-x <make-s 5 10>)

(STRUCT-select) e₁ → 5

(KONG) e → (f 5)

env=

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                     [true (+ x 1)])
                     [true x]))
(define c <make-s 5 10>)
```

(Prog)

Reduktion am Beispiel

(f 5)

e = (f 5)

(FUN) e → (cond [($<$ 5 1) (/ 5 0)]
[true (+ 5 1)]
[true 5])

env=

(define-struct s (x y))
(define (f x) (cond [($<$ x 1) (/ x 0)]
[true (+ x 1)])
[true x]))
(define c <make-s 5 10>)

(Prog)

Reduktion am Beispiel

(cond [(< 5 1) (/ 5 0)]

[true (+ 5 1)]
[true 5])

e = (cond [(< 5 1) (/ 5 0)] [true (+ 5 1)] [true 5])

E = (cond [] (/ 5 0)] [true (+ 5 1)] [true 5])

und $e_1 = (< 5 1)$

(PRIM) $e_1 \rightarrow \text{false}$

(KONG) e \rightarrow (cond [false (/ 5 0)] [true (+ 5 1)] [true 5])

env=

(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
[true (+ x 1)])
[true x]))

ake-s 5 10>)

(Prog)

Reduktion am Beispiel

(cond [false (/ 5 0)]

[true (+ 5 1)]

[true 5])

e = (cond [false (/ 5 0)] [true (+ 5 1)] [true 5])

(COND-False) e → (cond [true (+ 5 1)] [true 5])

env=

(define-struct s (x y))

(define (f x) (cond [(< x 1) (/ x 0)]
[true (+ x 1)])
[true x]))

c <make-s 5 10>

(Prog)

Reduktion am Beispiel

(cond [true (+ 5 1)]
[true 5])

e = (cond [true (+ 5 1)] [true 5])
(COND-True) e → (+ 5 1)

env=

(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
[true (+ x 1)])
[true x]))
c <make-s 5 10>)

(Prog)

Reduktion am Beispiel

(+ 5 1)

e = (+ 5 1)
(PRIM) e → 6

env=

(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
[true (+ x 1)])
[true x]))
c <make-s 5 10>)

(Prog)

Reduktion am Beispiel

6

```
env=
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                     [true (+ x 1)])
                     [true x]))
(define c <make-s 5 10>)
```

Refactoring durch Äquivalenzrelation

- Äquivalenzrelation:
 - 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 $e_2 \rightarrow^* e$
- Refactoring:
 - Änderung der Struktur des Programms ohne dass sich die Bedeutung verändert
 - Zur Verbesserung der Struktur
- Ersetzen eines Teilausdrucks durch äquivalenten Ausdruck
- Beispiel Kommutativität
 - $(\text{overlaps-circle } c1 \text{ } c2) \equiv (\text{overlaps-circle } c2 \text{ } c1)$
 - Refactoring: Vertauschen der Argumente

Refactoring durch Äquivalenzrelation

- Äquivalenzrelation bisher:
 - Funktionsaufrufe nur reduzierbar, wenn Argumente Werte sind
 - Unpraktisch für die Prüfung von Äquivalenz
- In BSL:
 - Auswertungsreihenfolge beeinflusst nicht Bedeutung des Programms
 - Funktionsaufruf
 - Auswertung der Teilausdrücke für Argumente nicht nötig vor Funktionsaufruf
 - Dann: Verwendung der Teilausdrücke im Funktions-Body

Allgemeinerer Auswertungskontext

$\langle E \rangle ::= '[]'$

| '()' <name? <e>* <E> <e>* ''
| ('cond' {[<e> <e>']}* '[' <E> <e> ']' {[<e> <e>']}* '')
| ('cond' {[<e> <e>']}* '[' <e> <E> ']' {[<e> <e>']}* '')
| ('and' <e>* <E> <e>* '')

- Gleiches darf überall mit Gleichem ersetzt werden
 - (*EKONG*): Falls $e_1 \equiv e_2$, dann $E[e_1] \equiv E[e_2]$
- Reduktion erhält Äquivalenz
 - (*ERED*): Falls $e_1 \rightarrow e_2$ dann $e_1 \equiv e_2$

Eigenschaften von Äquivalenzrelationen

- Reflexivität

- (*EREFL*): $e \equiv e$.

- Kommutativität

- (*EKOMM*): Falls $e_1 \equiv e_2$, dann $e_2 \equiv e_1$.

- Transitivität

- (*ETRANS*): Falls $e_1 \equiv e_2$ und $e_2 \equiv e_3$, dann $e_1 \equiv e_3$.

“Symbolische“ Auswertung

- Auswertung mit Platzhaltern
- Bestimmung von Werten für Argumente nicht nötig
- (EFUN): Falls (`define (name name1 ... namen) e`) in der Umgebung, dann
 $(\text{name } e_1 \dots e_n) \equiv e[\text{name}_1 := e_1 \dots \text{name}_n := e_n]$

Nutzung von Fachwissen über Äquivalenz

- Boolesche Ausdrücke
 - Konjunktion: Gesamtausdruck ist **false**, wenn mindestens ein Teilausdruck **false** ist
 $(\text{and} \dots \text{false} \dots) \equiv \text{false}$
 - Disjunktion analog
 - Weitere bekannte Äquivalenzen für boolesche Ausdrücke, z.B. De Morgansche Gesetze
- Arithmetik
 - Addition, Multiplikation, etc. sind kommutativ
 $(+ a b) \equiv (+ b a)$
 - Weitere bekannte Äquivalenzen für Arithmetik, z.B. Assoziativgesetz, Distributivgesetze
- Menge der Äquivalenzen für eingebaute Funktionen:
(EPRIM)

Äquivalenz und Resultat

Gegeben:

(define (f x) (f x))

(define (g x) 42)

(g (f 1))

Laut EFUN dürfen wir (g (f 1)) durch 42 ersetzen.

Es gilt:

(g (f 1)) ≡ 42

Laut Reduktionsregel FUN gilt jedoch $(g (f 1)) \rightarrow (g (f 1))$, da $(f 1) \rightarrow (f 1)$.

Gilt auch:

Der Ausdruck (g (f 1)) hat den Wert 42?

Die Berechnung terminiert also nicht und kann daher niemals den Wert 42 annehmen.

Nein! Weil die Berechnung des Ausdrucks nicht terminiert.

Äquivalenz und Resultat

- Es gilt also:
 - Falls $e_1 \equiv e_2$ und $e_1 \rightarrow^* v$, dann auch $e_2 \rightarrow^* v$, genau dann, wenn die Reduktion von e_1 und e_2 terminiert
- Eine schwächere Eigenschaft gilt:
 - Falls $e_1 \equiv e_2$ und $e_1 \rightarrow^* v_1$ und $e_2 \rightarrow^* v_2$, dann $v_1 = v_2$.