HASKELL

KAPITEL 1

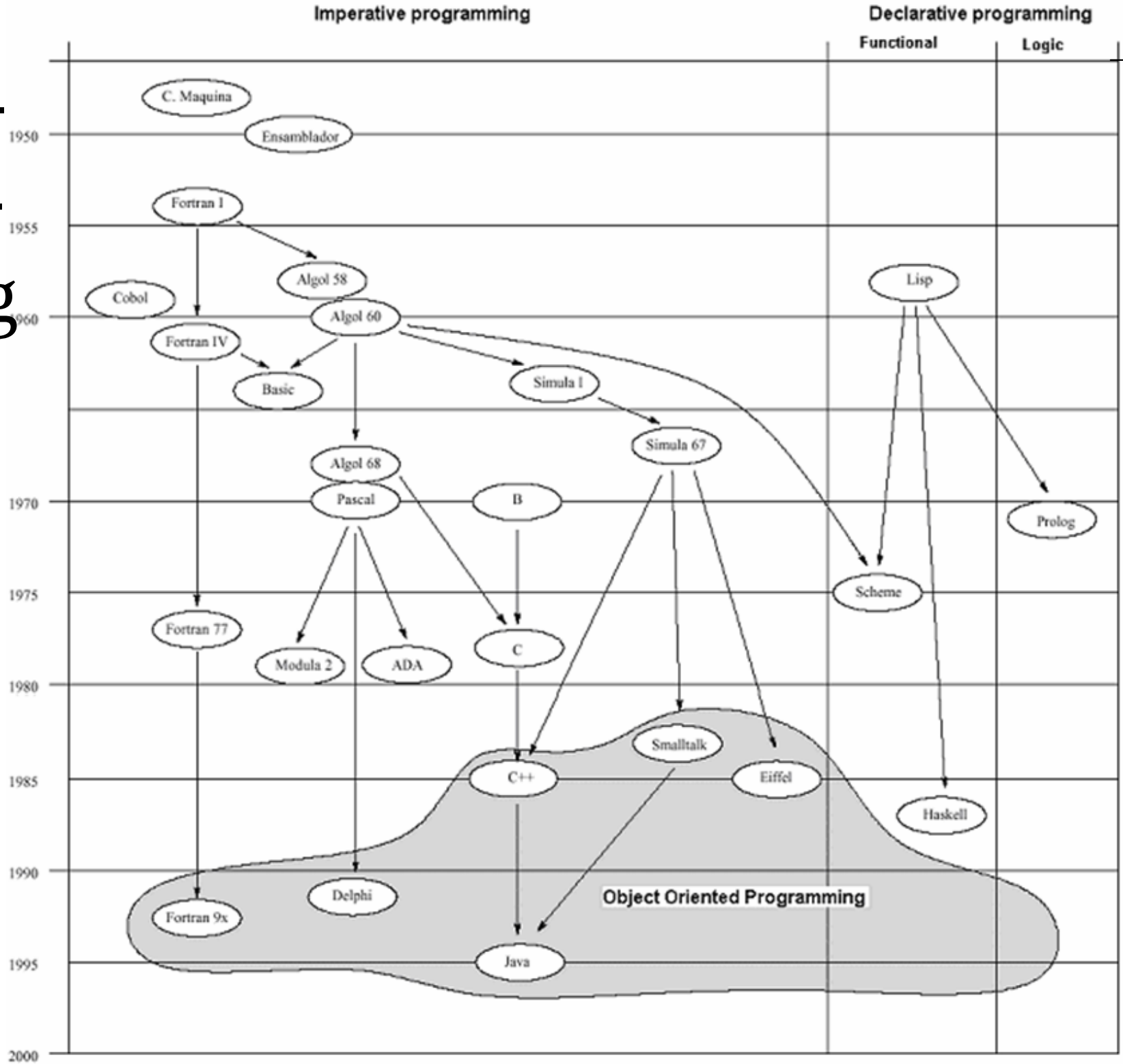
Fundamentale Konzepte

Richard Bird:
Introduction to Functional Programming using Haskell
2nd edition Prentice Hall (1998)

Simon Thompson
Haskell – The Craft of Functional Programming
Addison Wesley (1999)

Dank an: Prof. W. Reisig und Prof. Karsten Schmidt für Vorlesungsmaterialien

Ein-ord-nung


Imperative programming
Declarative programming
Functional
Logic
model based programming
1950
1955
1960
1970
1975
1980
1985
1990
1995
2000
C. Maquina
Ensamblador
Fortran I
Cobol
Algol 58
Algol 60
Fortran IV
Basic
Simula I
Simula 67
Algol 68
Pascal
B
Lisp
Prolog
Scheme
Fortran 77
Modula 2
ADA
C
Smalltalk
C++
Eiffel
Haskell
Delphi
Fortran 9x
Object Oriented Programming
Java
endliche Automaten
Petrinetze
SADT
Prozess-algebren
Statecharts
UML

http://www.haskell.org/aboutHaskell.html

About Haskell

Haskell is a computer programming language. In particular, it is a *polymorphi-caly typed*, *lazy*, *purely functional* language, quite different from most other programming languages. The language is named for Haskell Brooks Curry, whose work in mathematical logic serves as a foundation for functional languages. Haskell is based on *lambda calculus*, hence the lambda we use as a logo.

Why Use Haskell?

Writing large software systems that work is difficult and expensive. Maintaining those systems is even more difficult and expensive. Functional programming languages, such as Haskell, can make it easier and cheaper. For example, a new user who wrote a small relational DBMS in Haskell had this to say: WOW! I basically wrote this without testing; just thinking about my program in terms of transformations between types.

http://www.haskell.org/aboutHaskell.html

Haskell is a wide-spectrum language, suitable for a variety of applications. It is particularly suitable for programs which need to be highly modifiable and maintainable. Much of a software product's life is spent in *specification*, *design* and *maintenance*, and not in *programming*. Functional languages are superb for writing specifications which can actually be executed (and hence tested and debugged).

Such a specification then *is* the first prototype of the final program.

Functional programs are also relatively easy to maintain, because the code is shorter, clearer, and the rigorous control of side effects eliminates a huge class of unforseen interactions.

- **What is functional programming?**
- C, Java, Pascal, Ada, and so on, are all *imperative* languages. They are "imperative" in the sense that they consist of a sequence of commands, which are executed strictly one after the other. Haskell is a *functional* language. A functional program is a single expression, which is executed by evaluating the expression. Anyone who has used a spreadsheet has experience of functional programming.
- In a spreadsheet, one specifies the value of each cell in terms of the values of other cells. The focus is on *what* is to be computed, not *how* it should be computed. For example:
- we do not specify the order in which the cells should be calculated - instead we take it for granted that the spreadsheet will compute cells in an order which respects their dependencies.

was soll erreicht werden?

zwei Ziele vereinbaren:

1. ein Programm ist ein mathematisches Objekt.
   Jeder Aspekt ist sonnenklar .

2. ein Programm wird auf einem Rechner realisiert ...

Typische Probleme dabei: was passiert bei

- partiell definierter Operation *z.B. 1/x für x = 0*
- Typfehler *z.B. 2 + True*
- nicht abbrechender Schleife
- etc

dazu gehört ...

Umgang mit Datentypen

unendliche, endliche Typen

1.1 Grundidee des Funktionalen Programmierens

Beobachtung:

ein konventionelles Programm berechnet eine *Funktion*.

Vorschlag:

Programmierer formuliert Funktionen,

kombiniert eigene und vorgegebene Funktionen.

Wie hinschreiben?

als *Terme.*

Rechner „rechnet Terme aus“

Beispiele für das „Ausrechnen“ von Termen:

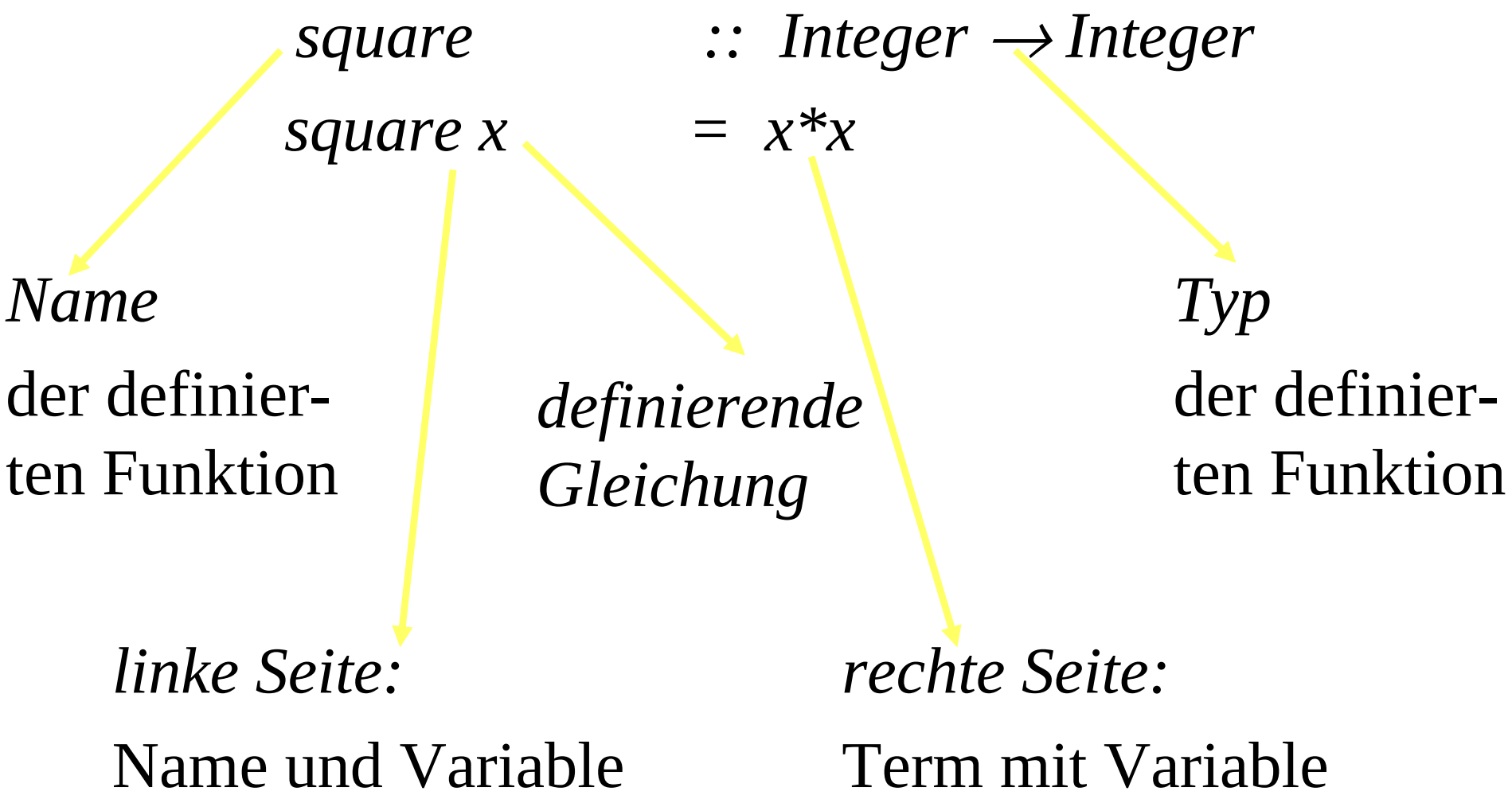
Term: *42*

ausrechnen: bleibt bei *42*

Term: *7*6*

ausrechnen: ergibt *42*

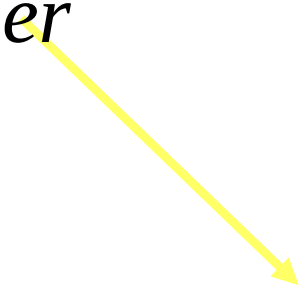
Beispiele für das Definieren einer Funktion



noch eine Definition einer Funktion

mathematisch übliche Notation:

$Integer \times Integer$



$smaller :: (Integer, Integer) \rightarrow Integer$

$smaller\ (x,y) = \textbf{if}\ x \leq y\ \textbf{then}\ x\ \textbf{else}\ y$

warum die neue Notation?

damit jede Funktion ***genau ein*** Argument hat

„Script“

square :: *Integer* → *Integer*

square x = *x*x*

smaller :: *(Integer, Integer)* → *Integer*

smaller (x,y) = **if** $x \leq y$ **then** *x* **else** *y*

Def.: ein *script* ist eine Menge von Definitionen.

Variante des Scripts

square :: *Integer* → *Integer*

square x = *x***x*

ersetze *Integer* durch *Float*:

Float: floating point numbers.

square :: *Float* → *Float*

square x = *x***x*

Variante des Scripts

square :: *Float* → *Float*

square x = *x***x*

kombinieren zu einem Script:

delta :: *(Float, Float, Float)* → *Float*

delta (a,b,c) = *sqrt(sqare* b – 4*a*c)

systemdefinierte Funktionen

1.2 Evaluation

$square :: Integer \rightarrow Integer$

$square\ x = x*x$

$smaller :: (Integer, Integer) \rightarrow Integer$

$smaller\ (x,y) = \textbf{if}\ x \leq y\ \textbf{then}\ x\ \textbf{else}\ y$

Beispiel für einen Term

square (smaller (5, 3+4))

„ausrechnen“: ergibt 25.

... geht das immer?

Kann man jeden Term „ausrechnen“?

was heißt „einen Term *t* ausrechnen“?

... den „einfachsten“ Term des gleichen Wertes angeben.

... gibt es immer so einen „einfachsten“ Term ?

Beispiel:	*square* (3+4)
= { Definition von + }	*square* (7)
= { Definition von *square* }	7*7
= { Definition von *}	49

alternativ:	*square* (3+4)
= { Definition von *square* }	(3+4)*(3+4)
= { Definition von + }	7*(3+4)
= { Definition von + }	7*7
= { Definition von *}	49

was heißt „einen Term t ausrechnen“?

... manchmal gibt es verschiedene Wege,

die aber zum gleichen Ergebnis führen.

... klappt das immer?

Beispiel

Skript:

three :: *Integer* → *Integer*

three x = *3*

infinity :: *Integer*

infinity = *infinity + 1*

einen Term ableiten:	*three infinity*
= { Definition von *infinity* }	*three* (*infinity* + 1)
= { Definition von *infinity* }	*three* ((*infinity* + 1) + 1)

etc. .....

Beispiel

Skript:

three :: *Integer* $\rightarrow$ *Integer*

three x = *3*

infinity :: *Integer*

infinity = *infinity + 1*

einen Term ableiten: *three infinity*

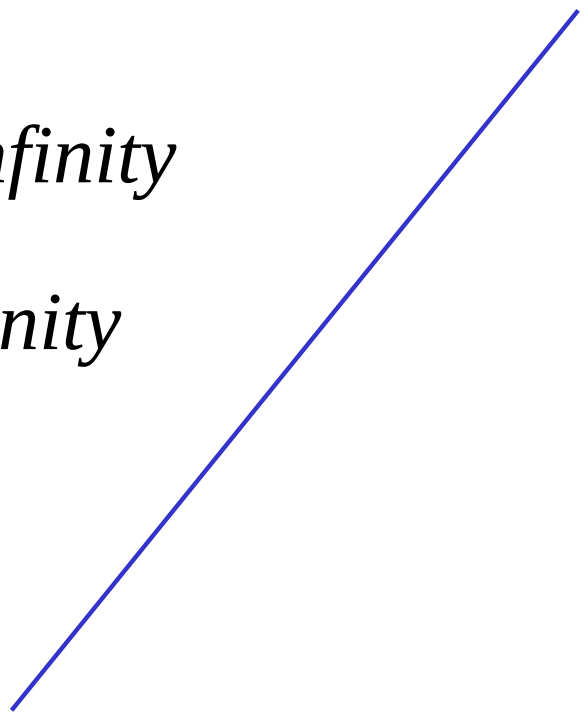
= { Definition von *three* } *3*

1.3 Werte

Was für Werte verwenden wir?	Gibt es kanonische Repräsentanten?
natürliche Zahlen	ja
floating point Zahlen	ja
Wahrheitswerte	(ja)
Symbole	ja
Symbolketten	ja
Tupel vorhandener Werte	ja
Funktionen über vorh. Werten	nein
Listen über vorhandener Werte	(ja)
selbst definierte Werte	ja
neue Operationen über Werten	nein

hat jeder Term eine klare Bedeutung?

Was passiert beim Ausrechnen von	semantisch	im Rechner
1/0	$\bot$	interrupt
square infinity	$\bot$	failure
three infinity	$\bot$	interrupt
	oder 3	oder 3



„*undefiniert*“

kein darstellbarer Wert in Haskell

1.4 „Funktion“

auch eine Funktion f: A $\rightarrow$ B ist ein Wert
(in der Menge der Funktionen).
Beispiele:

three :: *Integer* $\rightarrow$ *Integer*

square :: *Integer* $\rightarrow$ *Integer*

delta :: *(Float, Float, Float)* $\rightarrow$ *Float*

Notationen

auch eine Funktion $f: A \to B$ ist ein Wert.

Man muss sie hinschreiben können:

three :: *Integer* $\to$ *Integer*

square :: *Integer* $\to$ *Integer*

delta :: *(Float, Float, Float)* $\to$ *Float*

allgemein schreibt man:

$f: A \to B$.

Für $a \in A$ gilt also: $f(a) \in B$.

hingeschrieben: $f\ a$

Beispiele für Notationen

auch eine Funktion f: A $\rightarrow$ B ist ein Wert.

Man muss sie hinschreiben können:

three :: *Integer* $\rightarrow$ *Integer*

square :: *Integer* $\rightarrow$ *Integer*

delta :: *(Float, Float, Float)* $\rightarrow$ *Float*

square 3 *delta (3.14, .5, 0.0)* *square (3+4)*

Klammer nötig wg. Unterschied zu *square 3+4*

Anwendung von *f* auf *a* bindet stärker als alles andere.

Deshalb: *square (square 3)* und nicht *square square 3*

1.4.1 „Extensionalität“

Def.: Zwei Funktionen $f: A \to B$ und $g: C \to D$ sind *gleich*, wenn

$A = C$, $B = D$,

und für jedes $a \in A$ gilt: $f(a) = g(a)$.

„egal, wie f und g implementiert sind“.

Beispiel:

double :: Integer → Integer
double x = x+x

double‘ :: Integer → Integer
*double‘ x = 2*x*

Dann gilt: double = double‘

auch ein Wert ist eine Funktion

Def.: Sei A eine Menge.

A^n bezeichnet die n-Tupel $(a_1, \ldots, a_n)$ über A.

klar für n = 2, 3, ...

„1-Tupel“ $(a) \in A^1$ „dasselbe wie“ $a \in A$.

„0-Tupel“ $( ) \in A^0$. Die Menge A^0 hat ein Element.

eine Funktion $f : A^0 \to A$ beschreibt einen Wert, $f( )$.

damit:

zu jedem $a \in A$ gibt es genau eine Funktion f mit $f( ) = a$

Damit: jedes $a \in A$ „ist“ eine Funktion $f : A^0 \to A$

HASKELL

KAPITEL 2

Funktionen als Programmiersprache

die Idee

was ist die Bedeutung eines klassischen Programms?
eine Funktion (oder Relation) zwischen Ein-und Ausgabe.

Idee:
schreibe Funktionen hin!
vermeide das Konzept „Zustand“ !
nutze Mathematik !

! eine Denkweise – analog *PROLOG*

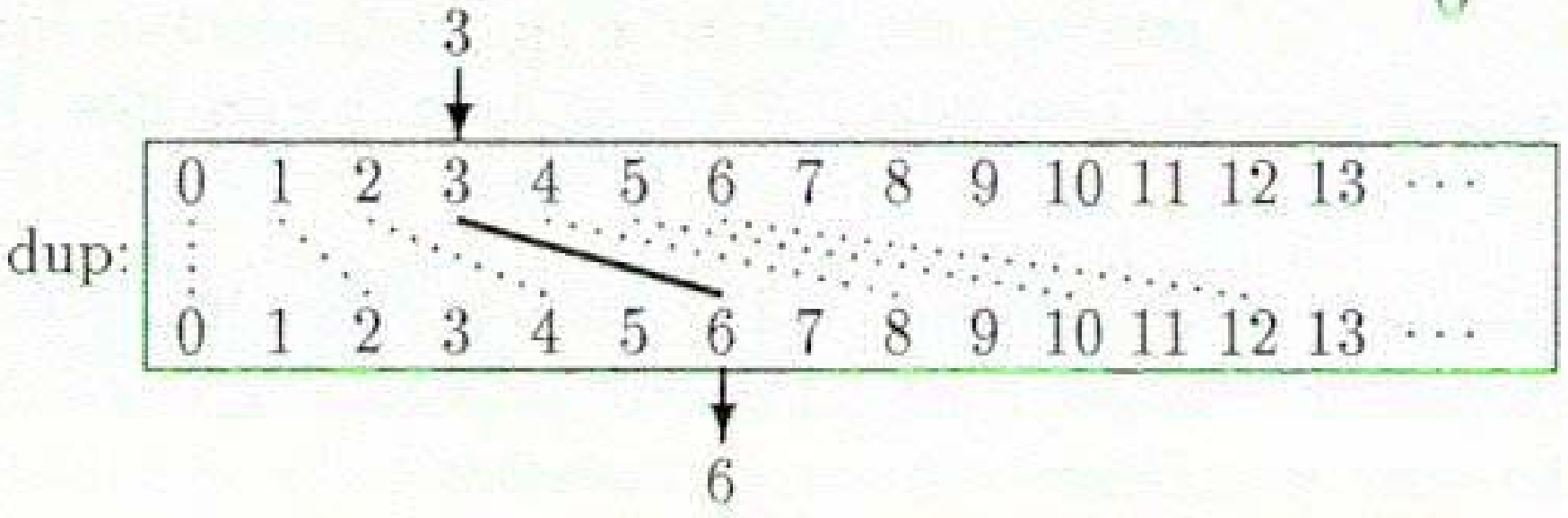
ein Programm beschreibt eine Funktion

Problem:

für f: A → B ist A (und B) i.a. unendlich.

deshalb nötig: eine Syntax, um solche Dinge hinzuschreiben.

Beispiel: Zahlen duplizieren. Abstrakt:
im Inneren des Kastens:



typische Dialoge

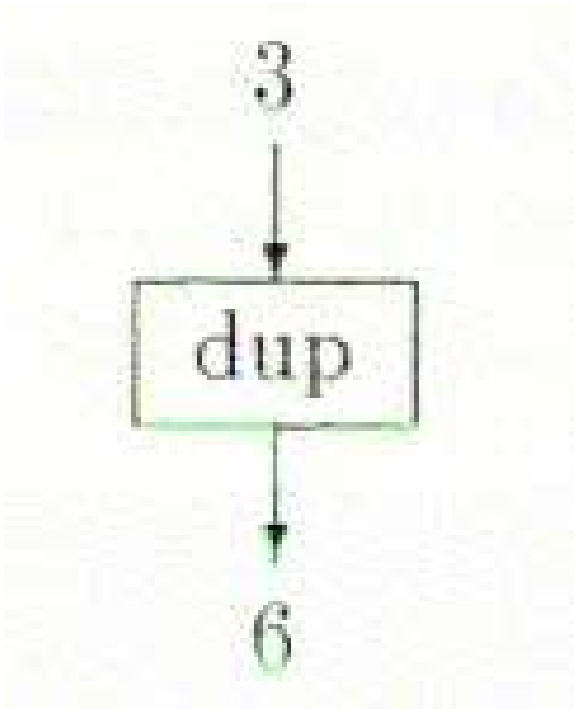
Beispiel

Eingabe: `dup 3`

Ausgabe: `>> 6`

Eingabe: `dup(3+1)`

Ausgabe: `>> 8`

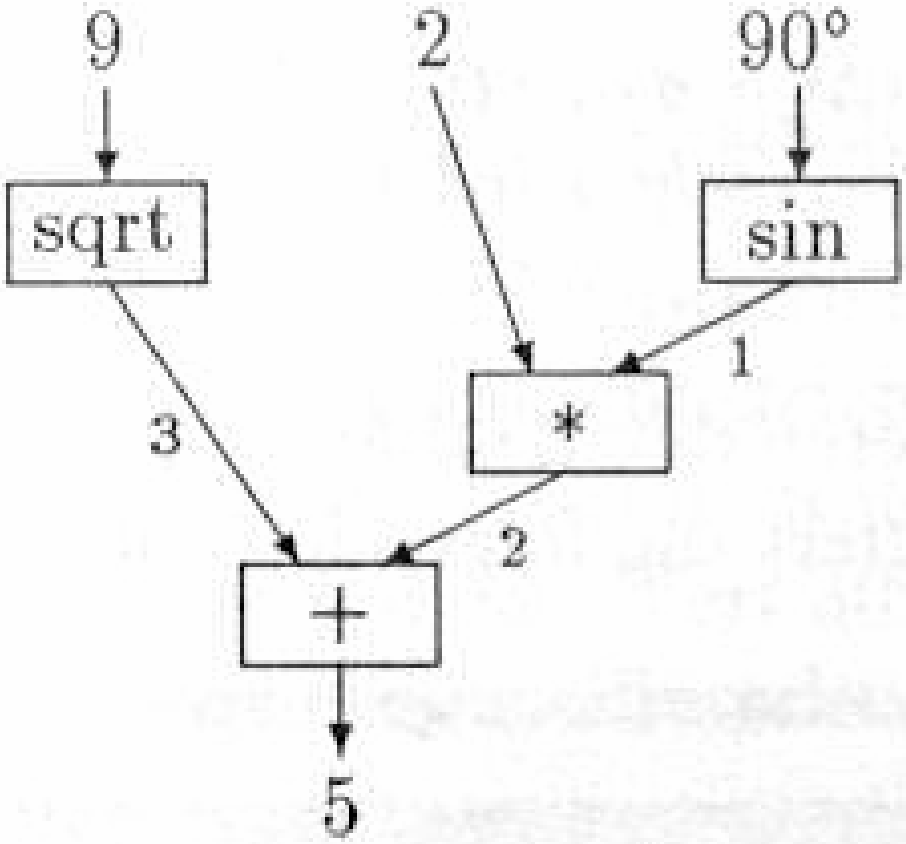


mehrere Funktionen verbinden

Eingabe: `sqrt(9) + (2*sin(90°))`

Ausgabe: `>> 5`

graphisch:



was brauchen wir?

elementare Ausdrücke (*Grundterme*):

- Konstanten `10` `9.81` `45`
- Funktionssymbole `sqrt, +, *, sin, °`
- zusammensetzen gemäß Stelligkeit

```
+ (sqrt(9), (* (2, sin(°(90)))))
```

schwer zu lesen, deshalb

- Infix für `+` und `*`
- Postfix für `°`
- Präfix nur für `sqrt`
- ergibt `sqrt(9) + (2*sin(90°))`

Beispiel:

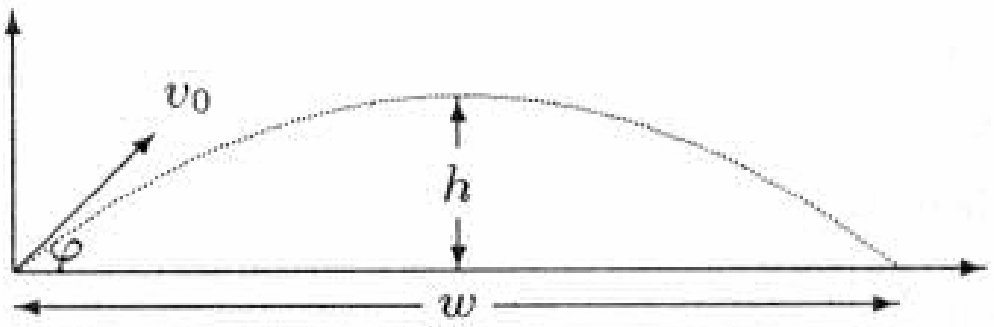


Abb. 2.1. Schiefer Wurf

Anfangsgeschwindigkeit: v_0

Anfangswinkel: φ

daraus ergibt sich die Wurfweite $w = \frac{v_0^2}{g} \sin 2\varphi$

Ausrechnen durch *Datenfluß*

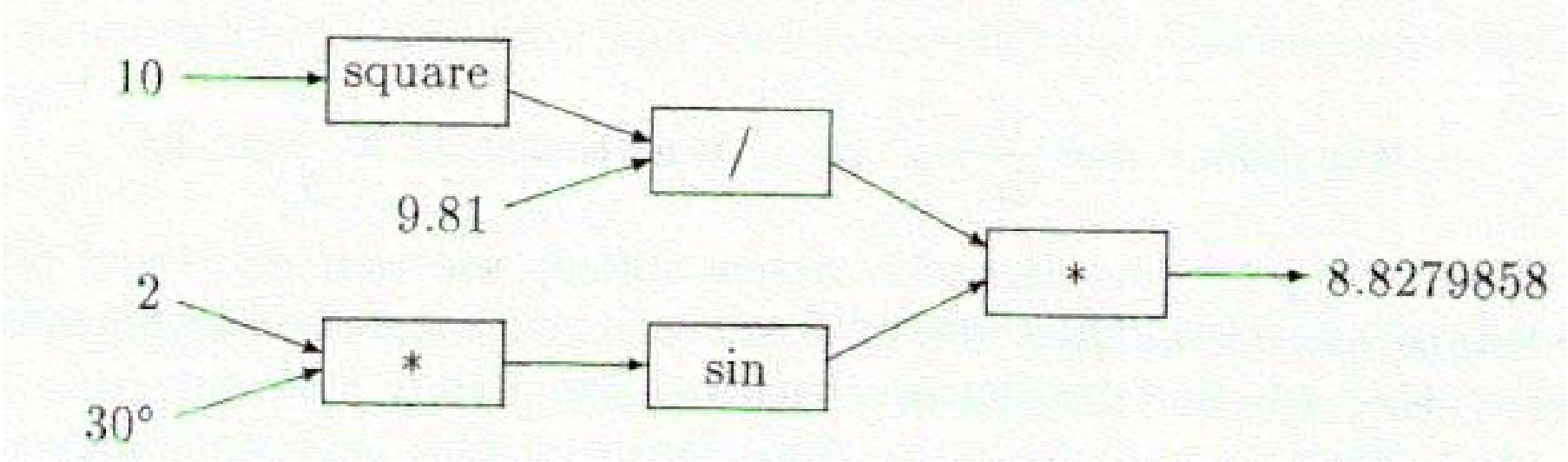
Beispiel:

Eingabe: `(square(10)/9.81)* sin(2*30°))`

Ausgabe: `>> 8.8279858`

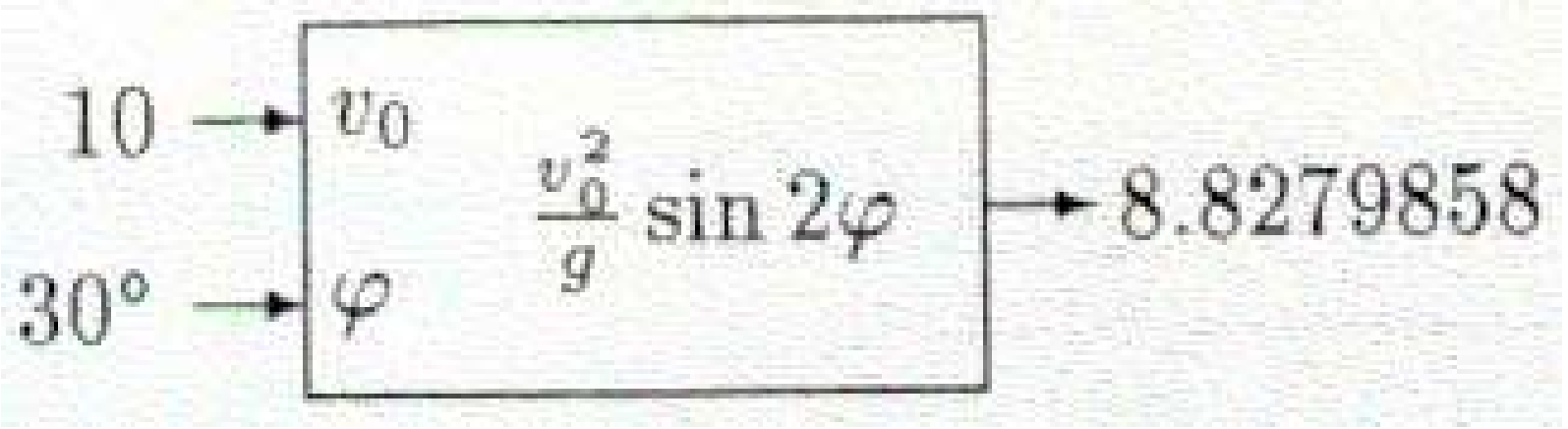
rechnet die Weite eines Wurfes aus, bei

- Anfangsgeschwindigkeit von 10 m/sec
- Winkel von 30°.



was wir eigentlich hinschreiben wollen

ein „Kästchen“



modifizierbar in den Inschriften der beiden linken Pfeile

... auf der Tastatur:

```
f (v0, phi) = (square(v0)/9.81) *
sin(2 * phi) (10, 30°)
```

Ausgabe: *>> 8.8279858*

„ausrechnen“ in 2 Schritten

```
f (v0, phi) = (square(v0)/9.81) *
sin(2 * phi) (10, 30°)
```

1. Schritt: das Funktionssymbol, die Variablenliste und die Argumente weglassen. Ergibt

```
(square(10)/9.81) * sin(2 * 30°)
```

allgemeine Form:

$f(x_1, \ldots, x_n) = \langle Ausdruck \rangle\ (v_1, \ldots, v_n)$

Ersetze in <*Ausdruck*> komponentenweise x_i durch v_i; lösche dann $f(x_1, \ldots, x_n)$ und $(v_1, \ldots, v_n)$.

Es entsteht ein Grundterm.

„ausrechnen“ in 2 Schritten

1. Schritt: Grundterm bilden

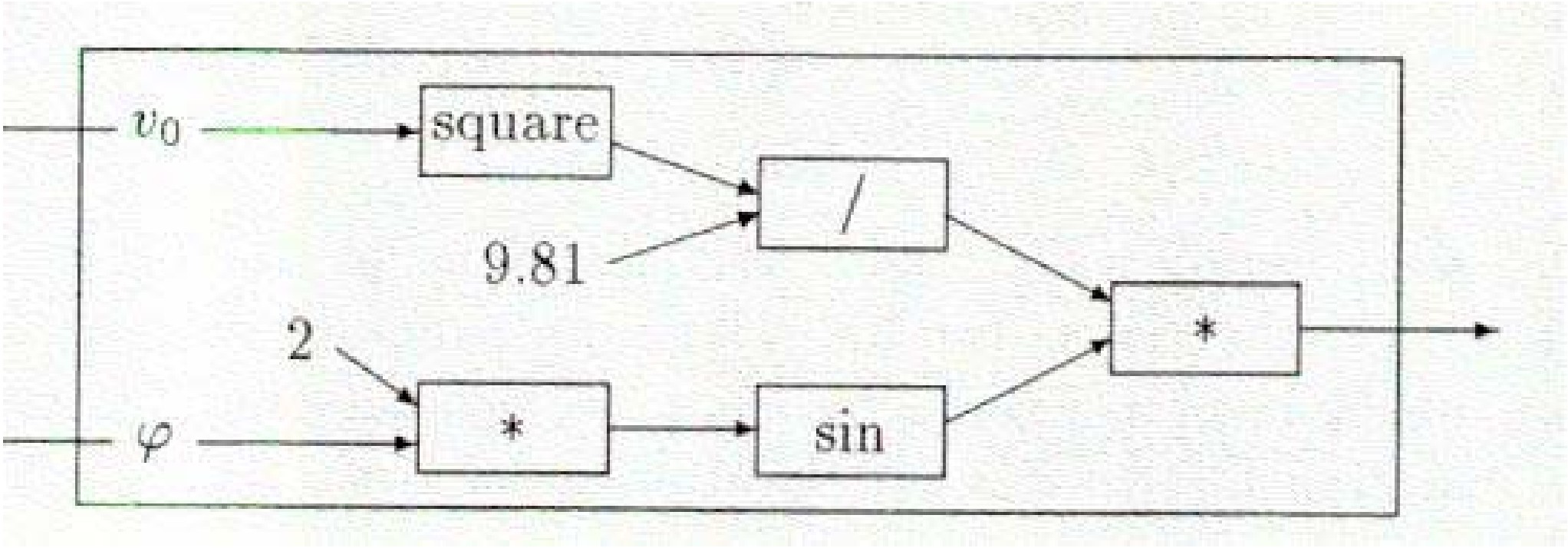
square((10)/9.81) * sin(2 * 30°)

2. Schritt: Grundterm berechnen

ergibt:

8.8279858

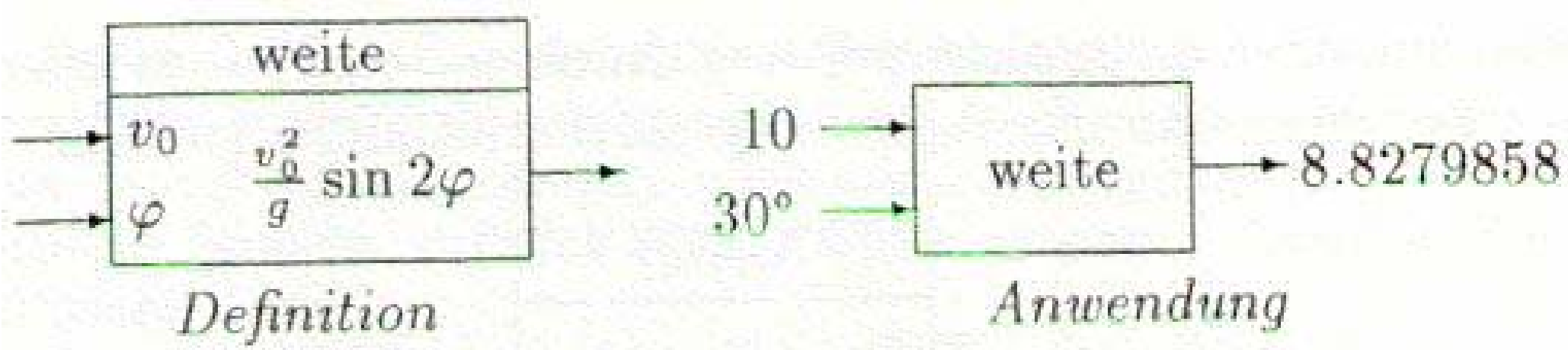
als Flussgraph:



Parameter: v_0 und φ

Konstante 9.81 und 2 innerhalb der Funktion

die Funktion *weite*



Funktionsname intuitiver:

```
weite (v0, phi) = (square(v0)/9.81) *
    sin(2 * phi)
```

Verwendung:

```
weite(10, 30°)    Ausgabe: >> 8.8279858
weite(10, 45°)    Ausgabe: >> 10.19368
weite(10, 30°)    Ausgabe: >> 8.8279858
```

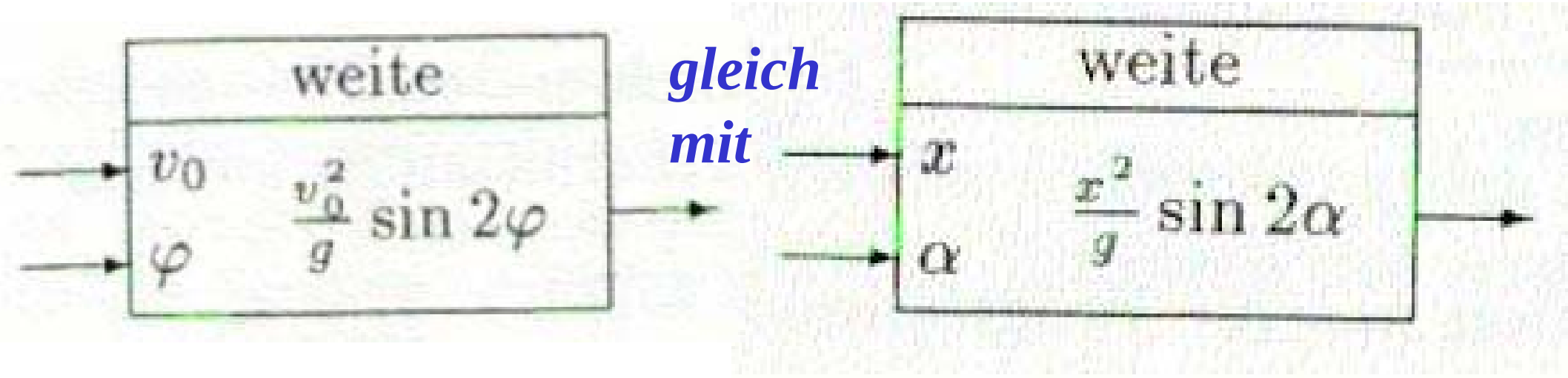
Variablen-Namen

... ohne Belang:

```
f   (v0, phi) = (square(v0)/9.81) * sin(2
    * phi))
```

genauso gut wie:

```
f   (x, alfa) = (square(x)/9.81) * sin(2
    * alfa))
```



... zunächst den Typ

... angeben ist zweckmäßig, aber in Haskell nicht zwingend:

```
mwstsatz :: Float
netto :: Float -> Float
mwstanteil :: Float -> Float
mwstsatz = 0.19
netto brutto = brutto-(brutto
*mwstsatz)
mwstanteil brutto = brutto
*mwstsatz
```

bisheriges

```
weite  :: (Float,Float)  ->  Float

weite (v0, phi) = (square(v0)/9.81) *
   sin(2 * phi) (10, 30°)
```