Currying

Um die Anzahl der Klammern in einem Ausdruck zu reduzieren, ersetzt man oft ein *Tupel* von Argumenten durch eine *Folge* von Argumenten.

Denn jeder Funktion des Typs

$$f : (X \times Y) \to Z$$

entspricht eineindeutig eine Funktion des Typs

$$curry(f) : X \to (Y \to Z)$$

Dann braucht man nur noch Funktionen mit *einem* (einfachen) Parameter betrachten und kann bekannte Theorien wie den λ-Kalkül verwenden.

Diese Idee von Moses Schönfinkel wurde von Haskell B. Curry populär gemacht und heißt nach ihm *currying*.

Funktion als Ergebnis

Betrachten wir die Additionsfunktion:
Natürlich können wir die Parameter zu *einem* Tupel zusammenfassen:
`add (x,y) = x + y` vom Typ `add :: (Int, Int) -> Int`

Die übliche Variante (ohne Klammern und Komma) ist aber folgende:
`add x y = x + y` von Typ `add :: Int -> (Int -> Int)`

Eine Funktionsanwendung `add e1 e2` ist linksassoziativ, also äquivalent zu `(add e1) e2`.

Die Anwendung der Funktion `add` auf *ein* Argument liefert eine neue Funktion für einen (den zweiten) Parameter.

Auf diese Weise können wir auch eine Inkrement-Operation definieren und erhalten so eine Funktion, die eine Funktion zurückliefert: `inc = add 1`

partielle Applikation

Damit können wir uns auf den Standpunkt stellen, dass jede Funktion nur *einen* Parameter hat.

Funktionen mit mehreren Parametern sehen wir als Funktionen höherer Stufe an:
add verarbeitet nur den ersten Parameter (**partielle Applikation**) und liefert als Ergebnis eine (namenlose) Funktion, die den zweiten Parameter verarbeiten kann.

Den Operator -> definieren wir dann als rechtsassoziativ:
der Typ `Int -> Int -> Int` ist also gleich `Int -> (Int -> Int)`.

Funktionen als Parameter

Zwei Funktionen, wo eine Funktion als Parameter vorkommt:

Beispiel

```
applyTwice f x = f (f x)      vom Typ
applyTwice :: (a -> a) -> a -> a
```

Die Klammern in der Typdeklaration sind notwendig, da der erste Parameter eine Funktion ist.
Der 2. Parameter hat –als Argument für die Funktion `f`– auch Typ `a`.

Beispiel

Die Funktion `iterate f x` liefert eine unendliche Liste von Werten zurück, die durch wiederholte Anwendung der Funktion `f` auf den Anfangswert `x` entsteht: `[x, f(x), f(f(x)), ...]`

```
iterate f x = x : iterate f (f x)      vom Typ
iterate :: (a -> a) -> a -> [a]
```

Anonyme Funktionen

Funktionen ohne Namen werden durch λ-Notation definiert:
Statt des Funktionsnamens wird ein λ vor die Parameter geschrieben.
In Haskell schreibt man statt λ einfach \ (*backslash*).

Beispiel

Eine anonyme Funktion, die drei Zahlen addiert: `\x y z -> x+y+z`

`dec x = x-1` ist also nur eine Abkürzung für die Definition des Namens
`dec = \x -> x-1`.

Anonyme Funktionen werden z.B. als Parameter benutzt.

Anonyme Funktionen

Beispiel (Nullstellenbestimmung durch das Newton-Verfahren)

Um die Quadratwurzel von z zu berechnen, bestimmen wir eine Nullstelle der Funktion $f(x) = x^2 - z$, indem wir den Grenzwert der Folge $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = \frac{1}{2}\left(x_i + \frac{z}{x_i}\right)$ mit $x_0 = 1$ berechnen.

Wir realisieren diese Folge durch eine unendliche Liste, da wir nicht wissen, wann der Grenzwert erreicht wird.

```
limes(a:b:x) = if a == b then a          -- Vorsicht: Rundungsfehler
               else limes(b:x)           -- beim Vergleich möglich

wurzel z =  limes (iterate g 1)
            where g(x) = (x + z/x)/2.0
```

Oder mit anonymer Funktion:

```
wurzel z = limes (iterate (\x -> (x + z/x)/2.0) 1)
```

Fakultät: bedingter Ausdruck

Ein **bedingter Ausdruck** ersetzt die bedingte Anweisung der imperativen Lösung:

```
fac n = if n == 0
        then 1
        else n * fac (n-1)
```

Fakultät: mehrere Gleichungen

Die Funktion wird durch **mehrere Gleichungen** definiert:

```
fac 0 = 1
fac n = n * fac (n-1)
```

Fakultät: zusätzlicher Parameter für Zwischenergebnis

Ein zusätzlicher Parameter a einer Hilfsfunktion `facAcc` speichert das Zwischenergebnis (Akkumulator) und wird mit dem neutralen Element der Multiplikation 1 initialisiert:

```
facAcc a 0 = a
facAcc a n = facAcc (n*a) (n-1)

fac = facAcc 1
```

Fakultät: zusätzlicher Parameter für Komposition der Funktionen

Ein zusätzlicher Parameter `k` einer Hilfsfunktion `facCps` merkt sich die noch zu erledigenden Funktionen und wird mit dem neutralen Element der Funktionskomposition `id` initialisiert:

```
facCps k 0 = k 1
facCps k n = facCps (k . (n *)) (n-1)

fac = facCps id
```

Hier bleiben nur endrekrusive Aufrufe übrig, die alle im gleichen Kellerrahmen abgearbeitet werden können (*continuation passing style*).

Fakultät: Listen

Mit Listen und Listenfunktionen:

```
fac n = product [1..n]
```

Bemerkungen zur Übersetzung von funktionalen Sprachen

Der auffälligste Unterschied von funktionalen gegenüber imperativen Sprachen ist, dass in funktionalen Sprachen **keine Namen** durch Zuweisung eines neuen Wertes **wiederverwendet** werden.

Übersetzung höherer Funktionen

Für die Berechnung eines Funktionswerts benötigt man

- den Code für die Funktion,
- die –innerhalb der Funktion vorkommenden– Werte der nicht-lokalen Variablen,
- die Werte der Parameter.

In einer Sprache erster Stufe ist eine Funktion eine Konstante.

Insbesondere ist die Anfangsadresse des Codes dieser Funktion zur Übersetzungszeit bekannt.

Übersetzung höherer Funktionen

In höher-stufigen Sprachen können *Werte* vorkommen, die selbst **Funktionen** sind.

Da dieser Wert –eine Funktion!– oft erst zur Laufzeit ermittelt wird, wird er durch einen **Funktionszeiger** (*code pointer*) implementiert, einen Zeiger auf den Code der Funktion, der bei einer Funktionsanwendung auszuführen ist.

Die Lebensdauer einer Funktion kann dadurch die Lebensdauer ihres Sichtbarkeitsbereichs überdauern!

Daher muss der Funktionszeiger mit allen von der Funktion benutzten Variablenbindungen auf der Halde abgelegt werden.

Übersetzung höherer Funktionen

Die first-class-Eigenschaft von Funktionen führt dazu, dass die Anzahl der Parameter von Funktionsdefinition und Funktionsaufruf nicht mehr übereinstimmen muss:

- **unterversorgte Funktionsaufrufe**: Enthält ein Funktionsaufruf weniger Parameter (z.B. 3) als die Funktionsdefinition (z.B. 5), so ist das Ergebnis eine Funktion mit den restlichen (hier: $5 - 3 = 2$) Parametern.
- **überversorgte Funktionsaufrufe**: Enthält ein Funktionsaufruf mehr Parameter (z.B. 5) als die Funktionsdefinition (z.B. 3), so muss das Ergebnis eine Funktion sein, die dann die restlichen (hier: 2) Parameter als Argumente konsumieren kann.

Striktheitsanalyse

Eine naive Implementierung der Bedarfsauswertung führt zu ineffizienten Programmen.

Dieser Aufwand kann stark verringert werden, wenn man für viele Ausdrücke bereits zur Übersetzungszeit feststellt, dass sie zur Berechnung des Programm-Ergebnisses *immer* benötigt werden und sie dann durch Call-by-Value (statt durch Call-by-Need) auswertet.

Eine **Striktheitsanalyse** (*strictness analysis*) soll diese Ausdrücke finden.

Striktheitsanalyse

Eine n-stellige Funktion f heißt strikt im i-ten Argument, falls der Wert dieses Arguments auf jeden Fall, also für alle möglichen Werte der Parameter von f, zur Berechnung des Funktionsergebnisses benötigt wird.

Wird zur Berechnung des Ergebnisses einer Funktion f der Wert eines Arguments sicher benötigt, terminiert jedoch die Berechnung dieses Arguments nicht, so terminiert die gesamte Berechnung nicht.

Stellt man einen undefinierten Wert –z.B. eine nicht-terminierende Berechnung– durch $\perp$ dar, wird Striktheit formal definiert als:

Eine n-stellige Funktion f heißt **strikt im i-ten Argument** $(1 \leq i \leq n)$ genau dann, wenn gilt: $x_i = \perp \Rightarrow f\ x_1 \ldots x_n = \perp$

Wird der Wert dieses Arguments auf jeden Fall benötigt, so kann der i-te aktuelle Parameter bei einer Anwendung der Funktion f schon beim Aufruf der Funktion f berechnet werden.

Striktheitsanalyse

Ähnlich kann man bei Bindungen in *nicht-rekursiven* `let`-Ausdrücken verfahren: Wird der Wert eines Bezeichners x aus der Definitionsliste eines `let`-Ausdrucks zur Berechnung des Wertes des `let`-Ausdrucks auf jeden Fall benötigt, so braucht die Berechnung der rechten Seite der Definition von x nicht verzögert werden.

Leider ist die Striktheit einer Funktion unentscheidbar;
jeder Algorithmus approximiert daher nur das Striktheitsverhalten.
Die Approximation muss aber „von der richtigen Seite“ kommen:

- Ist eine Funktion strikt, so darf das Analyseergebnis „nicht strikt“ lauten – dann wird Optimierungspotential verschenkt.
- Eine nicht-strikte Funktion darf aber nie als strikt klassifiziert werden – eine Optimierung könnte dann die Bedeutung des Programms verändern.

Eine Striktheitsanalyse nutzt daher vor allem bei Funktionen erster Ordnung und flachen Wertebereichen.

Übersetzung des Call-by-Need

Das Prinzip der **Bedarfsauswertung** (*lazy evaluation*) besteht darin, dass bei Funktionen die Auswertung der Argumente solange verzögert wird, bis die Auswertung nötig wird.

Es genügt, einen Ausdruck soweit zu berechnen, dass der äußerste Daten-Konstruktor des Ergebniswertes bestimmt werden kann (*weak head normal form*); Unterkomponenten des Wertes können noch unausgewertet sein.

Nur in Sprachen ohne Nebenwirkungen lässt sich der **Call-By-Need** aber ohne allzu großen Mehraufwand implementieren:

Man sorgt dafür, dass eine spätere Auswertung eines Ausdrucks den gleichen Wert liefert, indem man die Bindungen aller nicht-lokalen („freien“) Variablen (**Umgebung**), die zu seiner Berechnung nötig sind, in einem „**Abschluss**“ abspeichert.

Übersetzung des Call-by-Need

Da nicht bekannt ist, wann die Auswertung stattfindet, kann ein Abschluss nicht im Keller gespeichert werden, sondern kommt auf die Halde.
Damit müssen aber auch alle Daten, die in einem Abschluss enthalten sein können, auf die Halde!

Ein Wert, der eine Funktion ist, verhält sich wie eine aufgeschobene Berechnung: Wird der Wert auf seine Argumente angewendet, ist die Berechnung ausgeführt.
Insofern ist ein Abschluss wie eine Funktion ohne Parameter.

Übersetzung des Call-by-Need

Die *lazy*-Implementierung ersetzt einen Ausdruck nach seiner ersten Auswertung durch seinen Wert. Dazu gibt es zwei Strategien:

- Jeder Abschluss erhält eine **Markierung** „ausgewertet“ bzw. „unausgewertet“.

  Wird der Wert eines Ausdrucks benötigt, wird diese Markierung geprüft.
  Ist der Abschluss schon ausgewertet, wird sein Wert zurückgegeben, sonst wird die aufgeschobene Berechnung ausgeführt, der Wert des Ergebnisses gespeichert und die Markierung verändert.

  Diese Strategie benötigt zusätzlich Platz für die Markierung und Code für einen Test (`IF` ausgewertet `THEN` ...) bei jedem Zugriff.

Übersetzung des Call-by-Need

Die *lazy*-Implementierung ersetzt einen Ausdruck nach seiner ersten Auswertung durch seinen Wert. Dazu gibt es zwei Strategien:

- Der Code zur Auswertung des Abschlusses wird nach der Auswertung durch **Code** ersetzt, **der den Wert zurückliefert**.

  Ein Wert braucht daher –wie ein Abschluss– auch einen Funktionszeiger, der auf den auszuführenden Code zeigt.

  Hier braucht man für Werte zusätzlich Platz für den Funktionszeiger (statt Markierung) und Code für einen Rücksprung (statt Test).

  Diese Lösung ist flexibler, da man den Code noch variieren kann (z.B. direkten oder indirekten Zugriff auf den Wert; zusätzliches Laden des Daten-Konstruktors in ein Register, um den Mustervergleich zu beschleunigen).

Teil VIII

Programmierparadigmen: Logische Programmierung

Einführung in die logische Programmierung

Anfänge der logischen Programmierung sind in den 1960er Jahren zu finden, als man Programme entwickeln wollte, die automatisch Beweise zu vorgelegten Aussagen konstruieren.

In den 1980er Jahren spielte logische Programmierung eine wichtige Rolle beim Bau von Expertensystemen.

Heute wird sie in den Bereichen Computerlinguistik und Künstliche Intelligenz verwendet.

Einführung in die logische Programmierung

Logische Programme bekommen als Datenbasis Fakten (*fact*, *axiom*) und Regeln (*rule*).

Wird dem Programm dann eine Frage (*query*, *goal*) gestellt, so versucht das Programm festzustellen, ob sich die Frage aus der Datenbasis durch logische Schlüsse ableiten lässt oder nicht.

Beispiel

Datenbasis:
Ein Hund ist ein Säugetier.
Ein Mensch ist ein Säugetier.
Ein Hund hat keine Arme.
Ein Mensch hat zwei Arme.
Ein Säugetier hat vier Beine und keine Arme oder es hat zwei Beine und zwei Arme.

Fragen:

Hat ein Mensch zwei Beine?	$\rightarrow$	ja
Hat ein Hund zwei Beine?	$\rightarrow$	nein
Wie viel Beine hat ein Hund?	$\rightarrow$	4

- Nirgendwo wird erklärt, wie das Programm zu seiner Antwort kommt. Wie der Beweis durchzuführen ist, wird nicht beschrieben.
- Die Reihenfolge der Fakten und Regeln ist logisch nicht relevant.
- Programme, die derartige Schlussfolgerungen ziehen können, heißen Deduktionssysteme.
- Eine Programmiersprache, die nur das Ergebnis (**Was** ist gesucht?) beschreibt, aber nicht den Berechnungsweg (**Wie** wird es gefunden?), heißt nicht-prozedurale Programmiersprache.
  Anderes Beispiel: SQL

Logische Ausdrücke

Logische Ausdrücke werden aufgebaut aus Prädikaten, d.h. Funktionen, die in die Wahrheitswerte **true** und **false** abbilden.

Für die Argumente der Prädikate benötigt man außerdem Konstante (z.B. Zahlen) und Variable.

Prädikate können über boolesche Funktionen wie **and**, **or** und **not** verknüpft werden.

Es gibt zwei Quantifizierer,

- den Allquantor $\forall X$ (Für alle X gilt ...) und
- den Existenzquantor $\exists X$ (Es gibt ein X, für das gilt ...),
  die jeweils eine Variable einführen, die in den Prädikaten als Argument auftreten kann.

Zu diesen Konstruktionsregeln für logische Ausdrücke gehören weiterhin Schlussregeln, die angeben, wie man aus einer Menge von (wahren) Fakten und Regeln neue (wahre) Aussagen erzeugen kann.

Beispiel

Das obige Beispiel in Prädikat-Schreibweise.

```
säugetier(hund).
säugetier(mensch).
arme(hund,0).
arme(mensch,2).
```

Für alle X gilt:

```
säugetier(X) ⇒ (beine(X,4) and arme(X,0)) or
               (beine(X,2) and arme(X,2)).
```

Bemerkung

Dabei wird stets angenommen, dass die Prädikate für die angegebenen Argumente den Wert true liefern und für alle anderen, nicht mit Hilfe der Regeln herleitbaren Werte, den Wert false liefern (Closed World Assumption).

Die Fragen lauten jetzt:

`beine(hund,4)?`	$\rightarrow$	ja
`beine(hund,2)?`	$\rightarrow$	nein
Existiert ein X mit `beine(hund,X)?`	$\rightarrow$	X=4

Horn-Klauseln

Man beschränkt sich auf eine Teilmenge aller logischen Ausdrücke:

Horn-Klauseln haben die Form

$$a_1 \textbf{ and } a_2 \textbf{ and } \ldots \textbf{ and } a_n \Rightarrow b$$

wobei die a_i genügend „einfache“ Ausdrücke sein müssen
(sie dürfen z.B. kein **or** und keine Quantoren enthalten).

Man bezeichnet b auch als Kopf und
„a_1 **and** a_2 **and** $\ldots$ **and** a_n“ als Rumpf der Klausel.

Jede im Kopf auftretende Variable ist implizit mit einem Allquantor,
jede andere Variable mit einem Existenzquantor verbunden.

Beispiel

```
parent(X,Y) and parent(Y,Z) => grandparent(X,Z)
```

ist zu interpretieren als:

Für alle X und Z gilt:
  Wenn es ein Y gibt,
    so dass X Elternteil von Y ist und Y Elternteil von Z ist,
      dann ist X Großelternteil von Z.

Schreibweise in Prolog

Horn-Klauseln werden in Prolog „seitenvertauscht“ geschrieben.
In der Prolog-Schreibweise wird außerdem das boolesche **and** durch ein Komma ersetzt:

$$\texttt{b :- } a_1 \texttt{ , } a_2 \texttt{ , ... , } a_n\texttt{.}$$

Fakten werden durch Horn-Klauseln mit leerem Rumpf dargestellt.

Als Konvention beginnen in Prolog Variablen immer mit Großbuchstaben, während Prädikate und Symbole mit einem kleinen Buchstaben beginnen.

Beispiel

Das obige Beispiel umgeschrieben und in Prolog-Notation:

```
saeugetier(hund).
saeugetier(mensch).
arme(hund,0).
arme(mensch,2).
beine(X,4) :- saeugetier(X) , arme(X,0).
beine(X,2) :- saeugetier(X) , arme(X,2).
```

Prinzipielle Arbeitsweise eines Prolog-Systems:

- Ausgehend von der eingegebenen Frage (dem Ziel) wird in der Datenbasis nach einer Horn-Klausel mit passendem Kopf gesucht.
- Die Prämissen werden dann in den Rumpf des Ziels substituiert (Resolution).

  b :- a_1, ... a_{i-1}, a_i, a_{i+1}, ... a_n. und

  c :- d_1, ... d_k. mit $c = a_i$

  Daraus kann man die folgende Klausel erzeugen:

  b :- a_1, ... a_{i-1}, d_1, ... d_k, a_{i+1}, ... a_n.
- Vorher müssen die Parameter von a_i und c unifiziert werden.
- Danach werden die einzelnen Teile des Rumpfes (Teilziele) auf die gleiche Weise bearbeitet.
  Dies endet, wenn alle Teilziele Fakten der Datenbasis (also `true`) sind.
- Gelangt man in eine Sackgasse, z.B. weil ein Teilziel nicht weiter verarbeitet werden kann oder keine Unifikation möglich ist, so wird eine andere Alternative versucht (Backtracking).

Man sieht an diesem Beispiel, dass zwei Fragen bei der logischen Programmierung nicht festgelegt sind, nämlich:

1. In welcher Reihenfolge werden passende Regeln angewendet?
2. In welcher Reihenfolge werden die Teilziele bearbeitet?

In Prolog ist die Reihenfolge genau festgelegt:

1. Die Datenbasis wird in der Eingabereihenfolge durchsucht („von oben nach unten“)
2. Die Teilziele werden von links nach rechts abgearbeitet. Neue Teilziele werden sofort berücksichtigt.

Beispiel

Man betrachte folgende Fakten und Regeln

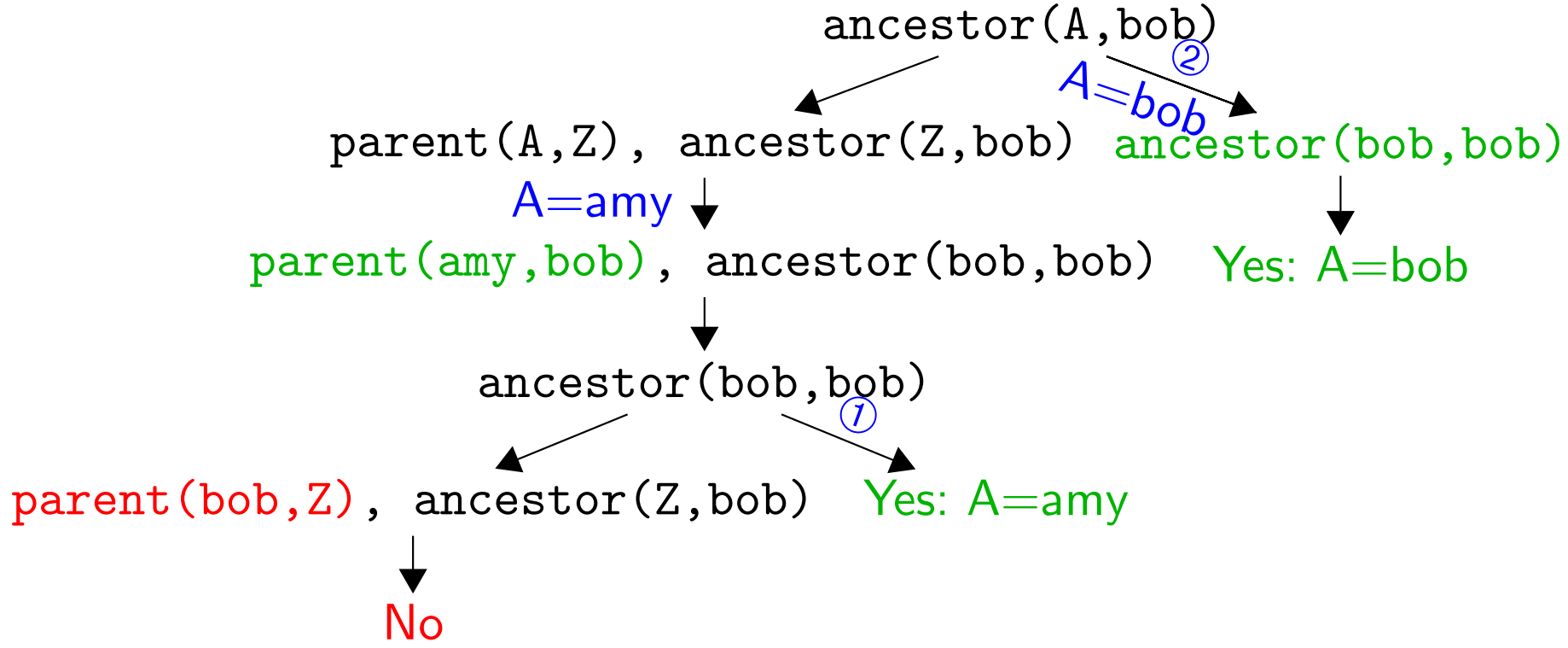
```
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
ancestor(X,X).
parent(amy,bob).
```

Ein ideales System liefert auf die Frage

```
?- ancestor(A,bob).
```

die Antworten `A = amy` und `A = bob`,
wobei die Reihenfolge der Antworten unbestimmt ist.

In Prolog ist die Berechnungsreihenfolge fest vorgegeben:
die Antworten erscheinen in fester Reihenfolge: `A=amy` vor `A=bob`.



Beispiel

```
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
ancestor(X,X).
parent(amy,bob).
```

Vertauscht man die beiden Teilziele in der ersten Regel:

```
ancestor(X,Y) :- ancestor(Z,Y) , parent(X,Z).
```

so erhält man bei Prolog keine Antwort mehr, da das System wegen Speichermangels stoppen wird.

Beispiel (Datenbasis: Regeln)

Eine Datenbasis für Verwandtschaftsbeziehungen:

```
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
sibling(X,Y) :- mother(M,X), mother(M,Y), X\=Y,
                father(F,X), father(F,Y).
cousin(X,Y) :- parent(U,X), parent(V,Y), sibling(U,V).
```

Beispiel (Datenbasis: Fakten)

```
father(bert, jeff).
mother(alice, jeff).
father(bert, george).
mother(alice, george).
father(john, mary).
mother(sue, mary).
father(george, cindy).
mother(mary, cindy).
father(george, vic).
mother(mary, vic).
```

Beispiel (Anfragen)

```
?- father(george,vic).
?- cousin(alice,john).
?- grandparent(bert,vic).
```