Generische Funktionen —
Datenabstraktion höherer Ordnung

- Generische Operationen
- Implementierung manifester Typen
- Datengesteuerte Programmierung
- Nachrichtenaustausch
- Typanpassung
- Typhierarchie
- Rekursive Datenabstraktion

Datenabstraktion höherer Ordnung

♣ Mehrere Repräsentationen eines abstrakten Datentyps sollen gleichzeitig verwendet werden können
(Komplexe Zahlen: Darstellung in Cartesischen und Polar-Koordinaten)

Mittel: *Generische Operatoren* — sollen Argumente in jeder eingeführten Darstellung verarbeiten.

Realisierung: Datenobjekte mit manifesten Typen — enthalten Information darüber, wie sie zu verarbeiten sind.

Daten-gesteuerte Programmierung als Implementierungsstrategie für Systeme generischer Operatoren

Horizontale und vertikale Strukturierung

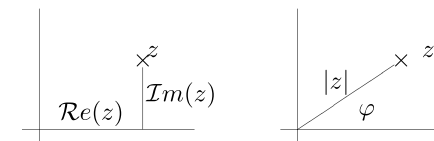
- Rein horizontale Struktur:

Programm, das komplexe Zahlen benutzt

Arithmetik komplexer Zahlen
`+c -c *c /c sqrt-c ...`
`=c print-c`

Impl. der komplexen Zahlen `make-compl real imag`

- Implementierung der komplexen Zahlen durch kartesische und durch Polarkoordinaten möglich

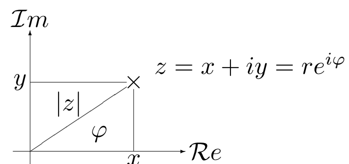


- Konsequenz:

<table>
<tr><td colspan="2">Programm,
das komplexe Zahlen benutzt</td></tr>
<tr><td colspan="2">Arithmetik komplexer Zahlen
<code>+c -c *c /c sqrt-c ...</code>
<code>=c print-c</code></td></tr>
<tr><td>Implem. der
komplexen Zahlen
in kart. Koord.
<code>make-rectangular</code>
<code>real-p. imag-p.</code></td><td>Implem. der
komplexen Zahlen
in Polarkoord.
<code>make-polar</code>
<code>magn. angle</code></td></tr>
</table>

Arithmethik komplexer Zahlen

- Zusammenhang:



- Cartesische Koordinaten: $z = x + iy \quad (i^2 = -1)$

  Polarkoordinaten: $z = re^{i\varphi}$

- Wir wählen immer die bequemere Darstellung!

- Addition und Subtraktion sind in kartesischen Koordinaten einfach zu implementieren:

$$\begin{aligned}\mathcal{R}e(z_1 + z_2) &= \mathcal{R}e(z_1) + \mathcal{R}e(z_2) \\ \mathcal{I}m(z_1 + z_2) &= \mathcal{I}m(z_1) + \mathcal{I}m(z_2)\end{aligned}$$

- Multiplikation und Division sind in Polarkoordinaten einfach zu implementieren:

$$\begin{aligned}\mathrm{abs}(z_1 \cdot z_2) &= \mathrm{abs}(z_1) \cdot \mathrm{abs}(z_2) \\ \arg(z_1 \cdot z_2) &= \arg(z_1) + \arg(z_2) \mod 2\pi\end{aligned}$$

- Zusammenhang zwischen beiden Darstellungen:

$$\begin{aligned}\mathcal{R}e(z) &= \mathrm{abs}(z) \cdot \cos(\arg(z)) \\ \mathcal{I}m(z) &= \mathrm{abs}(z) \cdot \sin(\arg(z)) \\ \mathrm{abs}(z) &= \sqrt{\mathcal{R}e(z)^2 + \mathcal{I}m(z)^2} \\ \arg(z) &= \arctan \frac{\mathcal{I}m(z)}{\mathcal{R}e(z)} \pm \text{ etwas mit } \pi\end{aligned}$$

  d.h. Umrechnung

  $x = r \cos \varphi$ $\quad$ $y = r \sin \varphi$

  $r = \sqrt{x^2 + y^2}$ $\quad$ $\varphi = \arctan y/x$

  `(atan y x)` liefert den Winkel, dessen Tangens y/x ist.

Cartesische und Polar-Darstellung

Beim Entwurf des Systems für komplexe Arithmetik folgen wir derselben Datenabstraktionsstrategie wie beim System für rationale Arithmetik.

- *Konstruktoren:*
  `(make-rectangular real-part imag-part)`
  `(make-polar magnitude angle)`
- *Selektoren:*
  `(real-part z)`, `(imag-part z)`
  `(magnitude z)`, `(angle z)`

GESETZE

```
z = (make-rectangular (real-part z) (imag-part z))
z = (make-polar (magnitude z) (angle z))
```


Arithmetische Operationen für komplexe Zahlen

```
(define (+c z1 z2)
  (make-rectangular
    (+ (real-part z1) (real-part z2))
    (+ (imag-part z1) (imag-part z2)) ))

(define (-c z1 z2)
  (make-rectangular
    (- (real-part z1) (real-part z2))
    (- (imag-part z1) (imag-part z2)) ))

(define (*c z1 z2)
  (make-polar
    (* (magnitude z1) (magnitude z2))
    (+ (angle z1) (angle z2)) ))
```



```
(define (/c z1 z2)
  (make-polar
    (/ (magnitude z1) (magnitude z2))
    (- (angle z1) (angle z2)) ))
```


Repräsentation komplexer Zahlen (1)

Nun: Repräsentation wählen; auf dieser Konstruktoren und Selektoren implementieren.

Variante 1:
Komplexe Zahlen als Paare (Realteil, Imaginärteil) — Cartesische Koordinaten

```
(define (make-rectangular x y) (cons x y))
(define (real-part z) (car z))
(define (imag-part z) (cdr z))
(define (make-polar r a)
  (cons (* r (cos a)) (* r (sin a)) ))
(define (magnitude z)
  (sqrt (+ (square (car z)) (square (cdr z)) )))
(define (angle z) (atan (cdr z) (car z)))
```

Repräsentation komplexer Zahlen (2)

Variante 2:
Komplexe Zahlen als Paare (Betrag, Winkel) — Polarkoordinaten

```
(define (make-rectangular x y)
  (cons (sqrt (+ (square x) (square y)))
        (atan y x)))
(define (real-part z)
  (* (car z) (cos (cdr z)) ))
(define (imag-part z)
  (* (car z) (sin (cdr z)) ))
(define (make-polar r a) (cons r a))
(define (magnitude z) (car z))
(define (angle z) (cdr z))
```


Problem multipler Repräsentationen

Die Operatoren sind unabhängig davon definiert, welche Repräsentation zugrundegelegt wird — Prinzip der Datenabstraktion!

Probleme treten auf, wenn beide Repräsentationsarten *gleichzeitig* (nebeneinander) verwendet werden.

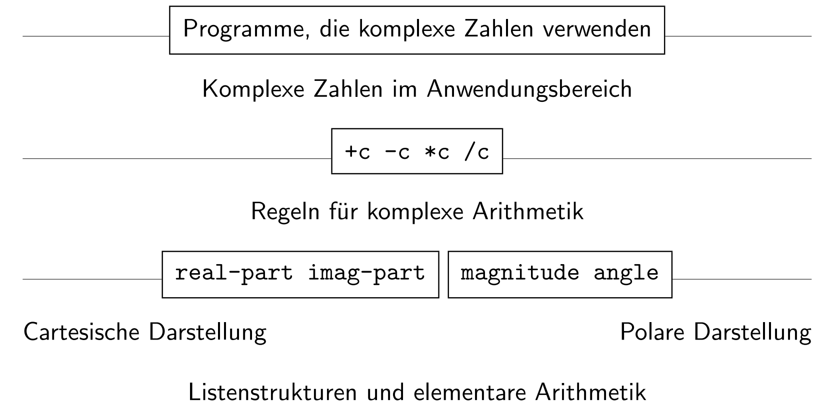
Sei das Paar `(3,4)` Darstellung einer komplexen Zahl. *Betrag = ?*

⇒ Cartesische Koordinaten: 5

⇒ Polar-Koordinaten: 3

Lösung des Problems: *Manifeste Typen*
Datenobjekte haben einen Typ, der überprüft werden kann.

Struktur des gener. Systems für komplexe Arithmetik



Implementation des Konzepts der manifesten Typen

Abstraktionsschritt:
Datenobjekte haben zwei Komponenten
`type` und `contents`

- Konstruktor:

```
(define (attach-type type-tag contents)
  (cons type-tag contents))
```

- Selektoren:

```
(define (type datum)
  (if (pair? datum)
      (car datum)
      (error "TYPE - bad typed datum" datum) ))
```

Implementation des Konzepts der manifesten Typen (2)

```
(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "CONTENTS - bad typed datum" datum) ))
```

- Typprüfungsprädikate:

```
(define (rectangular? z)
  (eq? (type z) 'rectangular))

(define (polar? z)
  (eq? (type z) 'polar))
```


Implementation des Konzepts der manifesten Typen (3)

Modifikation der Konstruktoren und Selektoren für komplexe Zahlen zur Berücksichtigung des Typs

```
(define (make-rectangular x y)
  (attach-type 'rectangular (cons x y)))
(define (make-polar r a)
  (attach-type 'polar (cons r a)))
```

Die Selektoren für getypte komplexe Zahlen werden mittels der Selektoren für die ungetypte Darstellung implementiert:
Der Typ wird zur Entscheidung für die Wahl der passenden Prozeduren benutzt.
⇒ **zwei Module** ("Pakete"):
- Verarbeitung der cartesischen Darstellung
- Verarbeitung der polaren Darstellung

Zwei Module: Selektoren (1)

```
(define (real-part z)
  (cond ((rectangular? z)
          (real-part-rectangular (contents z)) )
         ((polar? z)
          (real-part-polar (contents z)) )))
(define (imag-part z)
  (cond ((rectangular? z)
          (imag-part-rectangular (contents z)) )
         ((polar? z)
          (imag-part-polar (contents z)) )))
(define (magnitude z)
  (cond ((rectangular? z)
          (magnitude-rectangular (contents z)) )
         ((polar? z)
          (magnitude-polar (contents z)) )))
```



```
(define (angle z)
  (cond ((rectangular? z)
          (angle-rectangular (contents z)) )
         ((polar? z)
          (angle-polar (contents z)) )))
```

Selektoren (2)

Die spezifischen Selektoren werden übernommen, jedoch wegen Namens-Eindeutigkeit umbenannt.

Selektoren für cartesische Repräsentation

```
(define (real-part-rectangular z) (car z))

(define (imag-part-rectangular z) (cdr z))

(define (magnitude-rectangular z)
  (sqrt (+ (square (car z)) (square (cdr z)))))

(define (angle-rectangular z)
  (atan (cdr z) (car z)))
```


Selektoren (3)

Selektoren für polare Repräsentation

```
(define (real-part-polar z)
  (* (car z) (cos (cdr z)) ))

(define (imag-part-polar z)
  (* (car z) (sin (cdr z)) ))

(define (magnitude-polar z) (car z))

(define (angle-polar z) (cdr z))
```


Modularisierung

Das System für komplexe Arithmetik hat damit die abgebildete Struktur. Die drei Module *Komplexe Arithmetik, Cartesische Darstellung, Polare Darstellung* sind weitgehend voneinander unabhängig

⇒ Organisationsprinzip für die Implementierung derart strukturierter Systementwürfe:
Jeder Modul kann von anderen Personen/ Teams implementiert werden.
Gemeinsam: *Schnittstellen-Vereinbarungen*

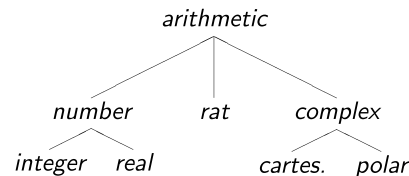
Jedes Objekt ist mit seinem Typ gekennzeichnet (*"tagged"*).
Daher können die Selektoren auf den Daten in generischer Weise operieren: Selektoren sind so implementiert, dass ihr "Verhalten" vom Typ des Datums bestimmt wird.

Generische Operationen

- $+$ ist für eine Vielzahl von Datentypen definiert:

Operandentyp	Funktion
ganzzahlig	Festpunktaddition
"reell"	Gleitpunktaddition
rational	`rat+`
komplex	`+c`
Vektoren	Vektoraddition
Matrizen	Matrixaddition
Polynome	Polynomaddition
...	...

- Programm kann immer + verwenden, verschiedene Pakete stellen unterschiedliche Implementierungen für die verschiedenen Datentypen bereit.
- Auswahl zur Übersetzungszeit: statische Typprüfung
- Auswahl zur Laufzeit: dynamische Typprüfung
- Typhierarchie (objektorientierte Programmierung!)



Allgemeine Strategie

- Der "Schnittstellen-Mechanismus" zwischen den Modulen ist so gestaltet, dass innerhalb eines gegebenen ("Repräsentations"-) Moduls (z.B. des polaren Moduls) eine komplexe Zahl ein ungetyptes Objekt ist: Paar (Betrag, Winkel)
- Wenn ein generischer Selektor auf ein Objekt des Typs `polar` zugreift, entfernt er den Typenindikator und übergibt den ungetypten Inhalt an den Polar-Modul
- Wenn eine komplexe Zahl konstruiert und aus dem polaren Modul "exportiert" wird, erhält es einen manifesten Typ, sodass es von den generischen Operatoren identifiziert werden kann.

Prinzipien der multiplen Repräsentation und generischen Operatoren

- Innerhalb eines Repräsentationstyps werden ungetypte Daten verwendet
- Generische Operatoren "steuern" über die Typen die entsprechenden (Sub-) Module an.

Schwachpunkte der Strategie:

1. Die generischen Schnittstellenprozeduren (`real-part, ...`) benötigen Information über die Repräsentationstypen.
   Lösung: Daten-gesteuerte Programmierung
2. Obwohl die Module unabhängig entwickelt werden können, muss Eindeutigkeit von Prozedurnamen sichergestellt werden.
   Lösung: Umgebungen als Kontexte, die die Bedeutung von Namen in Ausdrücken festlegen

Daten-gesteuerte Programmierung

Motivation:
Was ist zu tun, wenn ein weiterer Repräsentationstyp hinzukommt?

- Einführung eines neuen Typindikators
- Erweiterung der generischen Schnittstellen-Prozeduren (`real-part, imag-part, magnitude, angle`) um jeweils eine Klausel in den `COND`-Termen:
  Typprüfung und Zugriff auf Selektor

Große Anzahl verschiedener Repräsentationstypen und generischer Operatoren: weiterer Modularisierungsschritt!
Zuordnung von Operatoren zu Typen wird aus Programmcode herausgenommen und in Operatoren-Tabelle eingebracht.

Daten-gesteuerte Programmierung

benutzt Operatoren-Tabelle direkt: Schnittstelle wird durch eine Prozedur implementiert, die unter Zuordnung Operator–Typ passende Prozedur nachschlägt und auf Operanden anwendet.

Erweiterung um neuen Repräsentationstyp ⇒ Erweiterung der Operatoren-Tabelle

Implementation:
Voraussetzung: Prozeduren zur Tabellenverwaltung

`(put <type> <op> <item>)`
fügt `<item>` an der durch `<type>` und `<op>` bestimmten Position in Tabelle ein

`(get <type> <op>)`
liefert Tabelleneintrag an der Position `<type>,<op>`; kein Eintrag: `'()`

Aufbau der Tabelle

```
(put 'rectangular 'real-part real-part-rectangular)
(put 'rectangular 'imag-part imag-part-rectangular)
(put 'rectangular 'magnitude magnitude-rectangular)
(put 'rectangular 'angle angle-rectangular)
```

Analog für den Modul Polare Repräsentation:

```
(put 'polar 'real-part real-part-polar) ...
```

Implementation der Auswahlprozedur

```
(define (operate op obj)
  (let ((proc (get (type obj) op)))  ; Zugriff
    (if (not (null? proc))
        (proc (contents obj)) ; Prozedur-Anwendung
        (error "OPERATE --- Operator undefined for this type"
          (list op obj)) ))))
```


Definition der generischen Schnittstellen-Prozeduren

```
(define (real-part obj) (operate 'real-part obj))

(define (imag-part obj) (operate 'imag-part obj))

(define (magnitude obj) (operate 'magnitude obj))

(define (angle obj) (operate 'angle obj))
```

Keine Änderung erforderlich, wenn neuer Repräsentationstyp hinzukommt!
Die allgemeine Strategie
Typ-Prüfung und Aufruf einer geeigneten Prozedur
heißt auch *"dispatching on type"*

Übung: Daten-gesteuerte Variante des Ableitungsprogramms

Nachrichtenaustausch ("message passing")

Daten-gesteuerte Programmierung löst Problem der Erweiterbarkeit.
Noch offen: Verwendung gleicher Namen in mehreren Modulen

Vor Einführung der Operator-Tabelle war "dispatching on type" den Operatoren zugeordnet: Zeilenorientierte Sicht auf die Tabelle *(typorientiert)* — Operatoren sind "aktiv"

Alternative Sichtweise: Zuordnung der Entscheidung zu den Datenobjekten: "dispatching on operator names"
Spaltenorientierte Sicht auf die Tabelle *(objekt-orientiert)* — jedem "aktiven" Datenobjekt entspricht eine Spalte.

D.h.: Datenobjekte müssen als aktive Objekte — Prozeduren — repräsentiert werden, die als Argument den Operatornamen übergeben bekommen und dann die geeignete Prozedur ausführen

Konstruktor für Datenobjekte

```
(define (make-rectangular x y)
  (define (dispatch m)
    (cond ((eq? m 'real-part) x)
          ((eq? m 'imag-part) y)
          ((eq? m 'magnitude)
           (sqrt (+ (square x) (square y))))
          ((eq? m 'angle) (atan y x))
          (else
           (error "MAKE-RECTANGULAR --- unknown op" m)) ))
  dispatch)   ; prozeduraler Wert
```


Anwendung eines generischen Operators (“Rollentausch”)

`(define (operate op obj) (obj op))`

Der Name der Operation wird als “Nachricht” an das Datenobjekt gesandt, das die Nachricht entschlüsselt und die entsprechende Operation ausführt. (vgl. `kons, kar, kdr`)

Integration der Arithmetik ganzer und rationaler Zahlen

Benötigt werden Module für

- Integer-Arithmetik
- rationale Arithmetik

Konstruktoren, Selektoren, Transformatoren

Generische arithmetische Operatoren

Entwurf analog zum bisherigen Vorgehen:

- Generischer Additionsoperator add:

`+`	für “gewöhnliche”
`+rat`	für rationale
`+c`	für komplexe Zahlen

Modul für elementare Arithmetik

```
(define (+number x y)
  (make-number (+ x y)))
(define (-number x y)
  (make-number (- x y)))
(define (*number x y)
  (make-number (* x y)))
(define (/number x y)
  (make-number (/ x y)))
```

Konstruktor:

```
(define (make-number n)
  (attach-type 'number n))
```

Modul für elementare Arithmetik (2)

Verbindung der Operatoren im Modul mit den generischen Operatoren
`add, sub, mul, div`

```
(put 'number 'add +number)
(put 'number 'sub -number)
(put 'number 'mul *number)
(put 'number 'div /number)
```

Definition der generischen Operatoren:

```
(define (add x y) (operate-2 'add x y))
(define (sub x y) (operate-2 'sub x y))
(define (mul x y) (operate-2 'mul x y))
(define (div x y) (operate-2 'div x y))
```


Modul für elementare Arithmetik (3)

Auswahlprozedur für zwei Argumente ("type dispatching")

```
(define (operate-2 op arg1 arg2)
  (let ((t1 (type arg1)))
    (if (eq? t1 (type arg2))
        (let ((proc (get t1 op)))
          (if (not (null? proc))
              (proc (contents arg1)
                    (contents arg2))
              (error "OPERATE-2 --- operator undef. on type"
                (list op arg1 arg2)) ))
        (error "OPERATE-2 --- operands not of same type"
          (list op arg1 arg2)) )))
         ; hier besser: Typenanpassung!
```


Integration des Moduls für komplexe Arithmetik

Schnittstellen-Prozeduren

Konstruktor (Typenindikator für "Export")

```
(define (make-complex z) (attach-type 'complex z))
```

Operatoren für komplexe Arithmetik als Aufrufe der generischen Op.

```
(define (+complex z1 z2)
  (make-complex (+c z1 z2)))
(define (-complex z1 z2)
  (make-complex (-c z1 z2)))
(define (*complex z1 z2)
  (make-complex (*c z1 z2)))
(define (/complex z1 z2)
  (make-complex (/c z1 z2)))
```


Erweiterung der Operatoren-Tabelle

```
(put 'complex 'add +complex)
(put 'complex 'sub -complex)
(put 'complex 'mul *complex)
(put 'complex 'div /complex)
```

⇒ *Zwei-Ebenen-Typensystem*
z.B. `(make-complex (make-rectangular 3 4))`

Die Operatoren `real-part, imag-part, magnitude, angle` sind nur innerhalb des Moduls für komplexe Arithmetik erreichbar,
d.h. sind nur definiert für Daten der Typen `rectangular` und `polar`.

Sie können jedoch eine Ebene höher "exportiert" werden, sodass sie auf Objekte des Typs `complex` anwendbar sind und automatisch auf den richtigen Repräsentationstyp weitergeleitet werden:

```
(put 'complex 'real-part real-part)
(put 'complex 'imag-part imag-part)
(put 'complex 'magnitude magnitude)
(put 'complex 'angle angle)
```


Typenanpassung (1)

Typenanpassung: Kombination von Operanden verschiedener Typen

Operatoren wurden bisher so definiert, als wären Datentypen voneinander unabhängig, *aber* es gibt sinnvolle Operationen mit Zahlen verschiedener Typen, z.B. Addition von einfachen und komplexen Zahlen

Ziel: Einführung der neuen Operatoren in kontrollierter Weise, sodass Modulgrenzen bei Verknüpfung von Operanden verschiedener Typen nicht verletzt werden.

Erster Lösungsansatz
Dreidimensionale Operatoren-Tabelle: Gen.Op $\times$ Typ-Arg1 $\times$ Typ-Arg2
z.B. Additionsoperatoren `+number-complex`, `+rational-complex`, etc.
Bei n Typen werden i.a. n^2 verschiedene Versionen jedes generischen Operators benötigt $\Rightarrow$ unvertretbar hoher Aufwand!

Typenanpassung (2)

Idee: Ausnutzung der Struktur des Typensystems — Objekte eines Typs können als Objekte anderer Typen angesehen werden,
z.B. Integerzahl als komplexe Zahl mit Imaginärteil 0.
$\Rightarrow$ Transformation des Problems einer arith. Op. zwischen Integer- und komplexen Zahlen in arith. Op. zwischen komplexen Zahlen

Realisierung: Typenanpassungs-Prozeduren (*"coercion procedures"*), z.B.

```
(define (number->complex n)
  (make-complex
    (make-rectangular (contents n) 0)))
```

Installation in einer besonderen "Anpassungstabelle"
`(put-coercion 'number 'complex number->complex)`
Bemerkung: Tabelle kann nur teilweise besetzt sein

Erweiterung von `operate-2`

Operanden vom selben Typ: wie bisher
Operanden von verschiedenen Typen: Anpassung, falls möglich

```
(define (operate-2 op arg1 arg2)
  (let ((t1 (type arg1))
        (t2 (type arg2)))
    (if (eq? t1 t2)
        (let ((proc (get t1 op)))
          (if (not (null? proc))
              (proc (contents arg1) (contents arg2))
              (error "OPERATE-2 --- operator undef. on type"
                (list op arg1 arg2)) ))
        (let ((t1->t2 (get-coercion t1 t2))
              (t2->t1 (get-coercion t2 t1)))
```

```
(cond
  ((not (null? t1->t2))
   (operate-2 op (t1->t2 arg1) arg2))
  ((not (null? t2->t1))
   (operate-2 op arg1 (t2->t1 arg2)))
  (else
   (error "OPERATE-2 --- operands of different type"
     (list op arg1 arg2))))))))
```


Typenhierarchien (1)

Da hier erforderliche Typenanpassungen nur von den Typen selbst abhängen — und nicht vom anzuwendenden Operator —, ist nur eine Prozedur für jedes Typenpaar bereitzustellen.

Das Anpassungsschema ist noch nicht allgemein genug:
Auch wenn keiner der Operanden an den anderen Typ angepasst werden kann, ist es evtl. möglich, beide an einen dritten Typ anzupassen.

Typenhierarchien (2)

Ausnutzung "globaler" Struktur in der Beziehung von Typen untereinander

```
complex
  ^
  |
 real
  ^
  |
rational
  ^
  |
integer
```

Anpassung von `integer` an `complex` braucht nicht explizit definiert zu werden, sondern kann über Zwischenschritte entlang der Hierarchie erfolgen

Typenhierarchien (3)

Realisierung in `operate-2`:
Einführung eines Operators `raise` zur Typenanhebung um eine Stufe in der Hierarchie: Anpassung von Objekten an den nächsthöheren Typ.

• In der Typenhierarchie kann man die *Vererbung* von Eigenschaften vorsehen: Wird ein Operator für geg. Objekt nicht gefunden, so wird Objekt auf Supertyp angehoben und der Operator dort gesucht, z.B. `real-part` für `integer`

• Weiterer Vorteil der Typenhierarchie:
Einfachste Repräsentation für ein Objekt kann durch "Absenkung" leicht gefunden werden,
z.B. $2+3i \;+\; 4-3i \;=\; 6+0i \;=\; 6$

Anmerkung: Lineare Anordnung von Typen ist i.a. nicht immer möglich.

Bei mehreren Super- oder Subtypen zur Entscheidung über geeignete Anhebung oder Absenkung entsteht u.U. großer Suchaufwand im Typengraphen!

ANHANG

ZUR SELBSTÄNDIGEN NACHARBEIT

ANWENDUNGSBEISPIEL: Computer-Algebra

Die Manipulation algebraischer Ausdrücke stellt hohe Anforderungen an den Systementwurf wegen der Komplexität und Vielschichtigkeit der zu lösenden Probleme.

Wir betrachten im folgenden einige Aspekte näher am Beispiel der Manipulation von Polynomen.

Konstruktion algebraischer Ausdrücke

Anfang mit einer Menge primitiver Objekte, z.B. Konstanten und Variablen.

Aufbau komplexer Ausdrücke durch Verknüpfung mit algebraischen Operatoren, z.B. Addition, Multiplikation, . . .

Polynome

Wie bei anderen formalen Sprachen: *Bildung von Abstraktionen.*

Typische Abstraktionen sind hier: Linearkombination, Polynom, rationale Funktion, trigonometrische Funktion, . . .
Diese können als Verbund-"Typen" betrachtet werden.

So kann z.B. der Ausdruck

$$x^2 \sin(y^2 + 1) + x \cos 2y + \cos(y^3 - 2y^2)$$

als Polynom in x beschrieben werden, dessen Koeffizienten trigonometrische Funktionen von Polynomen in y und der Koeffizienten ganze Zahlen sind.

Arithmetik mit Polynomen

Was sind Polynome?

Polynome üblicherweise definiert relativ zu bestimmten Variablen, den Unbestimmten der Polynome.
Betrachten o.B.d.A. nur *univariate* Polynome (nur eine Unbestimmte).

Polynom: Summe von Termen.

Term: Koeffizient oder Potenz der Unbestimmten
oder Produkt aus beiden.

Koeffizient: algebraischer Ausdruck, der nicht von der Unbestimmten abhängt.

Arithmetik mit Polynomen (2)

Beispiele:

$$5x^2 + 3x + 7$$

ist ein einfaches Polynom in x,

$$(y^2 + 1)x^3 + (2y)x + 1$$

ist ein Polynom in x, dessen Koeffizienten Polynome in y sind.

Fragen:

- Ist das erste Polynom dasselbe wie $5y^2 + 3y + 7$?
  Ja, wenn wir es als mathematische Funktion ansehen;
  Nein, wenn wir nur seine syntaktische Form betrachten.

Arithmetik mit Polynomen (2)

- Das zweite Polynom ist algebraisch äquivalent zu einem Polynom in y, dessen Koeffizienten Polynome in x sind.

  Soll ein Computer-Algebrasystem dies erkennen?
- Ein Polynom kann auf verschiedene Weise dargestellt werden, z.B. als Produkt von Faktoren, als Menge von Wurzeln (falls univariat), als Liste von Werten an einer geg. Menge von Punkten.

Ansatz zum Systementwurf

Entwurfsentscheidung:
Polynom als syntaktische Form
(nicht: dahinterstehende mathematische Bedeutung)

Systementwurf: Prinzip der Datenabstraktion

Polynom besteht aus Variabler und Menge von Termen

Konstruktor: `make-polynomial`

Selektoren: `variable, term-list`

Polynom-Addition

```
(define (+poly p1 p2)
  (if (same-var? (variable p1) (variable p2))
      (make-polynomial
        (variable p1)
        (+terms (term-list p1)
                (term-list p2)))
      (error "+POLY --- Polynomials not in same variable"
        (list p1 p2))))
```


Systementwurf (2)

Polynom-Multiplikation

```
(define (*poly p1 p2)
  (if (same-var? (variable p1) (variable p2))
      (make-polynomial
        (variable p1)
        (*terms (term-list p1) (term-list p2)))
      (error "*POLY --- Polynomials not in same variable"
        (list p1 p2))))
```

Installation dieser Prozeduren im generischen Arithmetik-System
(daten-gesteuerte Programmierung)

```
(put 'polynomial 'add +poly)
(put 'polynomial 'mul *poly)
```


Handhabung von Term-Listen

Konstruktoren:	`the-empty-termlist`
	`adjoin-term`
Selektoren:	`first-term`
	⇒ Term höchster Ordnung
	`rest-terms`
Prädikat:	`empty-termlist?`
Selektoren für einen Term:	`order, coeff`

Addition von Termen

```
(define (+terms L1 L2)
  (cond
    ((empty-termlist? L1) L2)
    ((empty-termlist? L2) L1)
    (else
      (let ((t1 (first-term L1))
            (t2 (first-term L2)))
        (cond
          ((> (order t1) (order t2))
           (adjoin-term t1 (+terms (rest-terms L1) L2)))
          ((< (order t1) (order t2))
           (adjoin-term t2 (+terms L1 (rest-terms L2))))
          (else
```

```
        (adjoin-term
          (make-term (order t1)
            (add (coeff t1) (coeff t2)))  ; gener. add
          (+terms (rest-terms L1)
                  (rest-terms L2)))))))))
```


Multiplikation von Termen

```
(define (*terms L1 L2)
  (if (empty-termlist? L1)
      (the-empty-termlist)
      (+terms (*term-by-all-terms (first-term L1) L2)
              (*terms (rest-terms L1) L2))))
(define (*term-by-all-terms t1 L)
  (if (empty-termlist? L)
      (the-empty-termlist)
      (let ((t2 (first-term L)))
        (adjoin-term
          (make-term
            (+ (order t1) (order t2))
            (mul (coeff t1) (coeff t2))) ; gener.
          (*term-by-all-terms t1 (rest-terms L))))))
```


Zur Termverknüpfung

Wegen Verwendung der generischen Operatoren **add** und **mul** können alle Typen von Koeffizienten verarbeitet werden, die dem Paket für generische Arithmetik bekannt sind.

Mit Typenanpassung können auch verschiedene Koeffiziententypen (z.B. rationale, komplexe) in einem Polynom behandelt werden!
Rekursion erlaubt auch, Koeffizienten, die selbst Polynome (in einer anderen Variablen) sind, korrekt zu verarbeiten.

Darstellung von Term-Listen
Term-Liste: Liste von Koeffizienten, auf die über die Ordnung des Terms zugegriffen wird.
Prinzipiell ist jede Technik zur Darstellung von Mengen geeignet. Da jedoch `+terms` und `*terms` auf Term-Listen sequentiell in absteigender Ordnung zugreifen: Repräsentation durch geordnete Listen.

Dicht besetzte Polynome

Ein Polynom heißt *dicht besetzt,* wenn in $\sum a_i x^i$ die meisten $a_i \neq 0$, sonst heißt es *dünn besetzt.*

Beispiele:
$x^5 + 2x^4 + 3x^2 - 2x - 5$ ist dicht besetzt,
$x^{100} + 2x^2 + 1$ ist dünn besetzt.

Effizienteste Darstellung der Term-Liste dicht besetzter Polynome:
Liste ihrer Koeffizienten, z.B. `(1 2 0 3 -2 -5)`
Aber: höchst ineffizient für dünn besetzte Polynome!

Daher:
Wähle Darstellung durch Listen von Paaren `((Grad Koeffizient) ...)`,
z.B. `((100 1) (2 2) (0 1))`

Konstruktoren, Selektoren und Prädikate

```
(define (adjoin-term term term-list)
  (if (=zero? (coeff term))  ; gener.
      term-list
      (cons term term-list)))
(define (the-empty-termlist) '())
(define (first-term term-list) (car term-list))
(define (rest-terms term-list) (cdr term-list))
(define (empty-termlist? term-list) (null? term-list))

; Darstellung von Termen
(define (make-term order coeff)
  (list order coeff))
(define (order term) (car term))   ; Grad
(define (coeff term) (cadr term))  ; Koeffizient
```



```
; Darstellung von Polynomen
(define (make-polynomial variable term-list)
  (attach-type 'polynomial
               (cons variable term-list)))
(define (variable p) (car p))
(define (term-list p) (cdr p))
```