ABSTRAKTION

Zweck der Abstraktion

Definition

eine Abstraktion erlaubt, eine (Teil-) Berechnung zu definieren
(und zu benennen) um sie danach (wiederholt) anzuwenden

ein Mechanismus zur Strukturierung von Programmen

jede Abstraktion erweitert die Sprache um eine neue Operation

beleibig viele (Niveaus) der Abstraktion sind möglich

Abstraktionen erlauben unterschiedliche Sichtweisen

- Was soll gemacht werden? (Benutzersicht): Kopf der Abstraktion
- Wie ist es implementiert? (Implementierersicht) Rumpf

Arten von Abstraktion

Funktionsabstraktion abstrahiert von einem Ausdruck

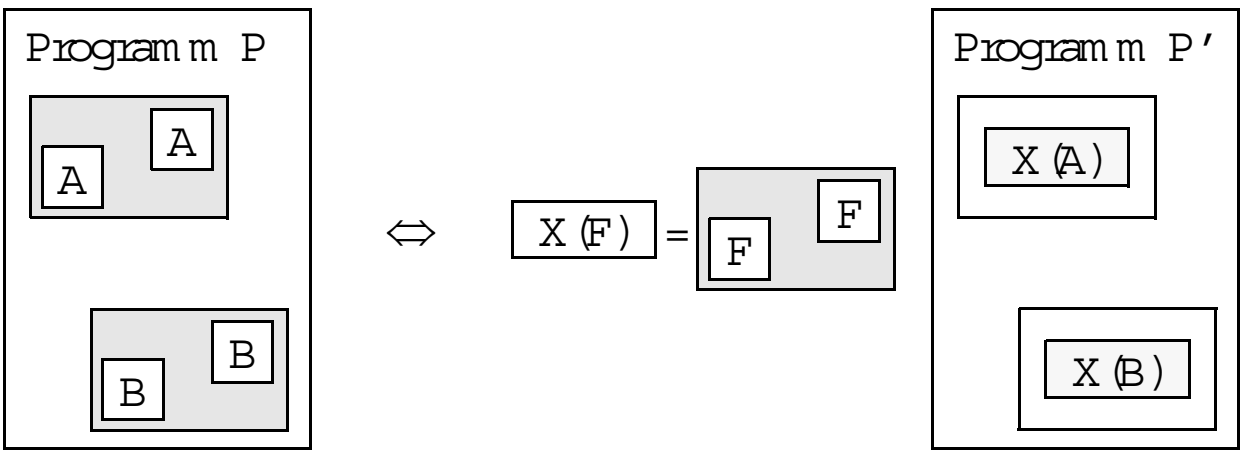
Prozedurabstraktion abstrahiert von einem Befehl

generische Abstraktion abstrahiert von einer Vereinbarung

...

Parametrisierung

erhöht die Flexibilität der Abstraktion



Funktionsabstraktion (Pascal)

eine Funktionsabstraktion

verkörpert einen auszuwertenden Ausdruck

der Benutzer nimmt nur das Ergebnis wahr (was),

nicht die Art und Weise der Berechnung (wie)

Beispiel: Pascal

Form: **function** I(F_1, ...; F_n) : T; B;

Rumpf: Befehlsfolge mit Zuweisungen I := V an den Funktionsnamen

```
function power (x: Real; n: Integer) : Real;
   begin (* Annahme: N >= 0 *)
   if n = 1 then power := x
   else power := x * power(x, n-1)
   end;
```

Nachteile:

Rumpf lädt zum Programmieren von Seiteneffekten ein

Es ist nicht einfach festzustellen,

daß immer mindestens eine Zuweisung I := V ausgeführt wird

Der Funktionsname wird mehrdeutig benutzt

```
power := x * power(x, n-1)
```

- für das Ergebnis der Funktion
- für rekursive Aufrufe, d.h. für die Funktion selber

Funktionsabstraktionen (ML)

Beispiel: ML

Kopf: **fun** I(F_1, ...; F_n) = B

Rumpf: ein Ausdruck (Ergebnistyp ist implizit)

```
fun power (x: Real; n: Integer) =
   if n = 1
   then x
   else x * power(x, n-1)
```

Benutzersicht:

`power` berechnet x^n (für positive `n`)

Implementiererersicht:

Bei jedem Aufruf von `power` wird der Algorithmus im Funktionsrumpf ausgeführt (rekursive Multiplikation)

Alternative Funktionsdefinition

```
fun power (x: Real; n: Integer) =
   if n = 1
   then x
   else if even(x)
      then power(sqr (x), n div 2)
      else power(sqr (x), n div 2) * x
```

Benutzersicht bleibt unverändert

Implementiererersicht:

- der neue Algorithmus benutzt Quadrieren und Multiplikation
- ist deshalb schneller

anonyme Funktionsabstraktionen (ML)

Funktionsabstraktion und Bindung sind zwei verschiedene Dinge

in Pascal geschieht das immer zusammen

in ML muß das nicht sein:

fn $(F_1, \ldots; F_n)$ => E Lambdaabstraktion

ist ein Ausdruck, der eine anonyme Funktionsabstraktion liefert

Beispiel "hoch 3"

Funktions-Ausdruck

```
fn (x: Real) => x * x * x
```

Funktions-Definition

```
val cube = fn (x: Real) => x * x * x
```

äquivalente Funktions-Definition

```
fun cube (x: Real) = x * x * x
```

Beispiel: Benutzung von `cube` in Ausdrücken:

Integral im Intervall [a,b]

```
fun integral (a: Real, a: Real, f: Real -> Real) = ...
```

Aufruf mit benannter Funktion

```
integral (0.0, 1.0, cube)
```

Aufruf mit anonymer Funktion

```
integral (0.0, 1.0, fn (x: Real) => x * x * x)
```

Prozedurabstraktionen

eine Prozedurabstraktion

verkörpert einen Befehl, dessen Ausführung Variablen verändert

der Benutzer nimmt nur diese Veränderungen wahr (was),

nicht die Art und Weise der Berechnung (wie)

Beispiel: Pascal

Form: **procedure** I(F$_1$,...;F$_n$);C;

```
type Words = array [ ... ] of Word;
procedure sort (var w: : Words);
    ...
```

Benutzersicht:

`sort(dict)` sortiert die Wörter in `dict` lexikographisch

Implementierersicht:

Bei jedem Aufruf von `sort` wird der Algorithmus im Prozedurrumpf ausgeführt (...)

Das Abstraktionsprinzip

Definition

für jede Art von Programmstück, das Berechnungen beschreibt,
kann eine Abstraktion gebildet werden

Was für Abstraktionen kann es noch geben?

Selektorabstraktionen

generische Abstraktion

Selektorabstraktion

Definition

Eine Selektor-Abstraktion abstrahiert von einem Variablen-Zugriff;
ein Selektor-Aufruf liefert den Verweis auf eine Variable

Beispiel (Pascal)

Schlangen ganzer Zahlen

```
type Queue = ...
function first (q : Queue) : Integer;
    ... (* liefert die erste Zahl in q *)
```

dies erlaubt den Aufruf

```
i := first(queue2);
```

aber nicht:

```
first(queue2) := first(queue2) - 1;
```

Beispiel (hypothetische Erweiterung von Pascal)

```
selector first (q : Queue) : var Integer;
    ... (* liefert Verweis auf die erste Zahl in q *)
```

dies erlaubt auch:

```
first(queue2) := first(queue2) - 1;
```

Selektorabstraktion zur Datenabstraktion

Beispiel

1. Schlangen als Felder

```
type Queue =record
       items : array [1.. max] of Integer;
       front, rear, count : 0.. max;
   end;
selector first (q : Queue) : var Integer;
   begin
      first := q.item[q.front]
   end;
```

2. Schlangen als Listen

```
type Queue = ^record
       head : Integer;
       tail : Queue;
   end;
selector first (q : Queue) : var Integer;
   begin
      first := q^.head
   end;
```

Fazit

Selektor-Abstraktionen sind nützlich

für das Programmieren abstrakter Datentypen

generische Abstraktion

Definition

eine generische Abstraktion abstrahiert von einer Vereinbarung

(z.B. von Prozeduren, Funktionen, Moduln)

Der Rumpf einer generischen Abstraktion ist eine Vereinbarung

Ihr Aufruf liefert eine Vereinbarung, durch Abarbeiten ihres Rumpfes.

Ihre Parameter können auch Typen sein!

Mehr dazu unter Kapselung

Parameter

Parameter machen Abstraktionen flexibler

```
function circumference (r : Real) : Real;
   begin
      circumference := 2 * pi * r
   end;
circumference (1.0) … circumference (a+b)
```

formaler Parameter

Bezeichner für ein Argument im Rumpf der Abstraktion (z.B. `r`)

aktueller Parameter

Ausdruck (o.ä.) der ein Argument liefert (z.B. `1.0` oder `a+b`)

Argument

eine Größe, die als Parameter an eine Abstraktion übergeben wird

Welche Werte können Argumente sein?

Pascal: primitive, zusammengesetzte Werte (außer Dateien),
Verweise auf Variablen, Prozedur- und Funktionsabstraktionen

ML: primitive, zusammengesetzte Werte, Funktionsabstraktionen

Parameterübergabe

Mechanismus, um die Korrespondenz zwischen aktuellen
und formalen Parametern herzustellen

Parameterarten

- Wertparameter (call by value)
- Variablenparameter (call by reference)
- Konstantenparameter
- Ergebnisparameter (**out**)
- Wert-Ergebnisparameter (**in out**)
- Funktionsparameter
- Prozedurparameter
- Namensparameter (Algol-60)

zwei Mechanismen erklären alle diese Parameter

- Kopiermechanismus
- definitorischer Mechanismus

Kopiermechanismen

Prinzip

Der formale Parameter F ist eine lokale Variable

das Argument A wird vor Ausführung der Abstraktion an F zugewiesen
und/oder
der Wert von F wird nach Ausführung der Abstraktion an das Argument
A zugewiesen (dann muß A eine Vraiable sein)

Wertparameter (call-by-value)

vor der Ausführung wird F angelegt
und das Argument (ein Wert) an x zugewiesen
während der Ausführung kann F gelesen und überschrieben werden
(Überschreiben wird aber nicht global wirksam)

Resultatparameter (call-by-result)

vor der Ausführung wird F angelegt, bleibt aber undefiniert
während der Ausführung kann F gelesen und überschrieben werden
nach Beendung der Ausführung wird der Wert von x dem Argument
(einer Variablen) zugewiesen

Wert/Ergebnisparameter (call-by-value-result)

vor der Ausführung wird F angelegt
und das Argument (der Wert einer Variablen) an x zugewiesen
während der Ausführung kann F gelesen und überschrieben werden
nach Beendung der Ausführung wird der Wert von F dem Argument
(einer Variablen) zugewiesen

Beispiel: Kopiermechanismen

Vektorrechnung in Ada

```
type Vector is array (1..n) of Real;
var a, b, c: Vector;
procedure add(val v, w : Vector;
        res sum : Vector) is
   begin
   for i in range 1..n do sum(i) := v(i) + w(i)
   end do
end add;
procedure normalize (val res u : Vector) is
   var s : Real;

   begin
   s := 0;
   for i in range 1..n do s := s + u(i) * u(i)
   end do; s := sqrt (s);
   for i in range 1..n do u(i) := u(i) / s
   end do;
end normalize;
```

ein Aufruf ...	bewirkt	... folgende Ausführung
add(a,b,c)	⇒	*v, w, sum: Vector;* *v := a; w := b;* **for** i **in range** 1..n **do** sum(i) := v(i) + w(i) **end do**; *c := sum*
...		
normalize (c)	⇒	*u: Vector := c;* s := 0; **for** i **in range** 1..n **do** s := s + u(i) * u(i) **end do**; s := sqrt (s); **for** i **in range** 1..n **do** u(i) := u(i) / s **end do**; *c := u*

definitorische Mechanismen

Prinzip

Der aktuelle Parameter wird an den formalen Parameter gebunden

Das funktioniert gleichermaßen für

- Konstantenparameter
- Variablenparameter
- Prozedur- und Funktionsparameter

Definition

vor der Ausführung werden die formalen Parameter als Synonyme (aliases) für die aktuellen Parameter definiert

während der Ausführung greift jeder Zugriff auf den formalen Parameter indirekt auf den aktuellen Parameter zu

Beispiel: definitorische Mechanismen

Vektorrechnung in Ada

```
type Vector is array (1..n) of Real;
var a, b, c: Vector;
procedure add(const v, w : Vector;
        var   sum : Vector) is
   begin
   for i in range 1..n do sum(i) := v(i) + w(i)
   end do
   end add;
procedure normalize (var u : Vector) is
   var s : Real;
   begin
   s := 0;
   for i in range 1..n do s := s + u(i) * u(i)
   end do; s := sqrt (s);
   for i in range 1..n do u(i) := u(i) / s
   end do;
end normalize;
```

ein Aufruf ...	bewirkt ...	folgende Ausführung
`add(a,b,c);`	⇒	*v: constant Vector = a;* *w: constant Vector = b;* *sum : Vector renames b;* `for i in range 1..n` `do sum(i) := v(i) + w(i)` `end do;`
`normalize(c)`	⇒	*u : Vector renames c;* `s := 0;` `for i in range 1..n` `do s := s + u(i) * u(i)` `end do; s := sqrt (s);` `for i in range 1..n` `do u(i) := u(i) / s` `end do;`

definitorische Mechanismen und aliasing

Beispiel: Pascal)

```
var i, j : Integer;
    a: array [0..10] of Integer;
procedure confuse (var m, n : Integer);
   begin
   n := 1; n := m + n
   end;
i := 100;
confuse(i, i);
…
read(j);
confuse(a[i], a[j])
```

Aliasing

- erschwert das Verstehen von Programmen
- läßt sich im allgemeinen nicht (algorithmisch) überprüfen

Vergleich der Mechanismen

Kopiermechanismen

- ... funktionieren nur auf Argumenten mit Zuweisung
  (meistens also nicht für Funktionen und Prozeduren)
- ... verlangen das Anlegen und Kopieren der Parameter
- ... synchronisieren die Übergabe
  (ein Zugriff vor/nach Ausführung des Rumpfes)

definitorische Mechanismen

- ... sind universell für alle Arten von Parametern
- ... machen das Kopieren der Argumente unnötig
  (sogar für Konstanten, die als Referenzen übergeben werden)
- ... synchronisieren die Übergabe nicht (aliasing)
  (jeder Zugriff bei der Ausführung
  greift erneut auf das Argument zu)

Zusammenfassung

Kopieren: ineffizient, aber sicher (insbesondere bei Nebenläufigkeit)
Definitorisch: Universell und effizient, aber nicht so sicher
(wenn es Seiteneffekte gibt)

Das Korrespondenzprinzip

Beobachtung

manche Parameterm echanismen entsprechen Vereinbarungen

- Konstantenparameter entsprechen Konstantendefinitionen
- Variablenparameter entsprechen Variablenumbenennungen
- Wertparameter entsprechen
  (initialisierenden) Variableneinführungen

Das Korrespondenzprinzip

Jeder Form von Vereinbarung entspricht

genau ein Parameterm echanismus und umgekehrt

ML beachtet das Korrespondenzprinzip (für Werte erster Klasse)

der definitorische Parameterm echanismus

entspricht der Wertdefinition

und wie ist es in Pascal?

Auswertungsreihenfolge

Einschränkung (zunächstmal)

reine funktionale Sprachen ohne Seiteneffekte

Beispiel (Pseudo-Haskell)

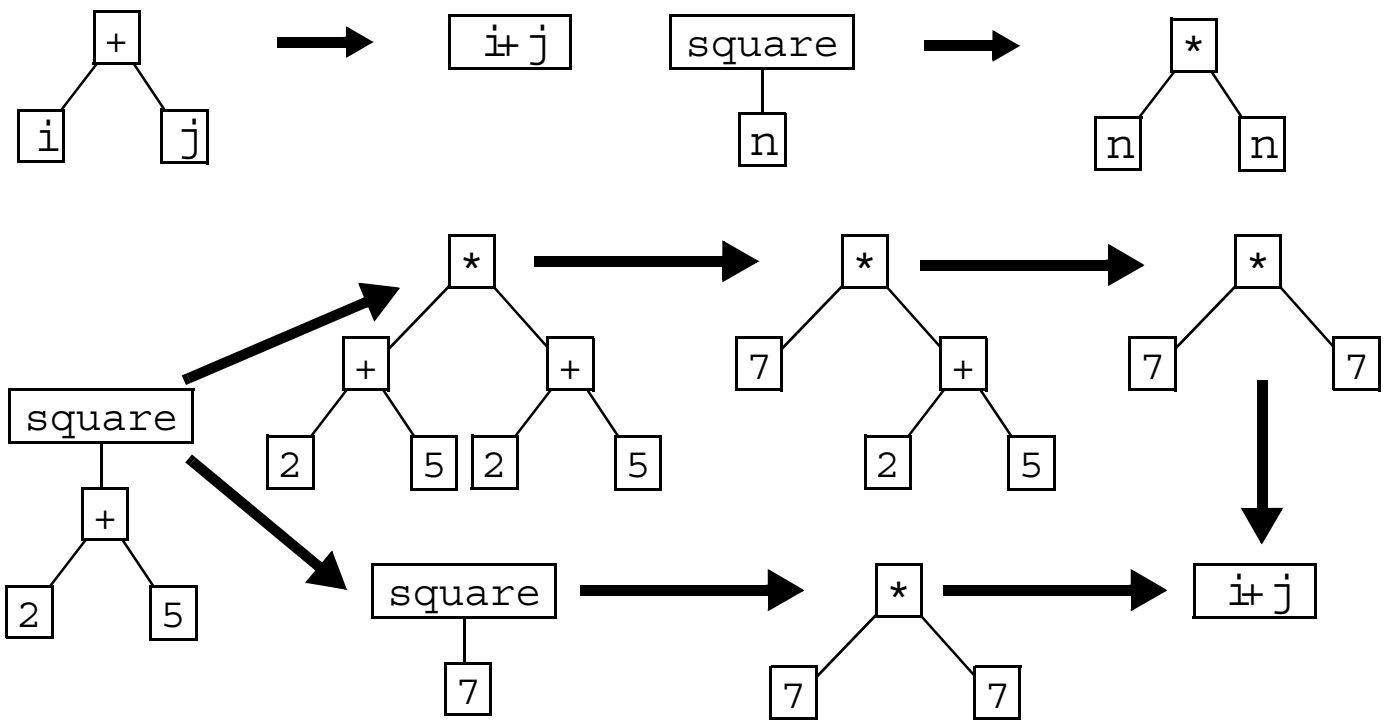
```
square(2+5)
where square n = n * n
      i + j = m where m ist die Summe von i und j
```

das Auswertungsmodell

Termersetzung (oder Lambdakalkül)

- Programmgleichungen werden als Baumersetzungsregeln aufgefaßt.
  (ersetze die linke Seite durch die rechte)
- sie werden auf den auszuwertenden Ausdruck angewendet ...
- ... bis keine Regel mehr anwendbar ist (... zur Normalform)

Beispiel



Auswertungsstrategien

allgemeine Auswertung

ist mehrdeutig

- an mehreren Stellen kann eine Regel angewendet werden
- verschiedene Regeln könnten angewendet werden
  (in funktionalen Sprachen ausgeschlossen)

Church-Rosser-Eigenschaft (auch Konfluenz)

alle erfolgreichen Ersetzungen liefern dasselbe Ergebnis

Strategien

beschränken die Auswahl der Stellen und Regeln

- Regeln von oben nach unten probieren
- äußerste Stelle
- innerste Stelle

normalisierende Strategien

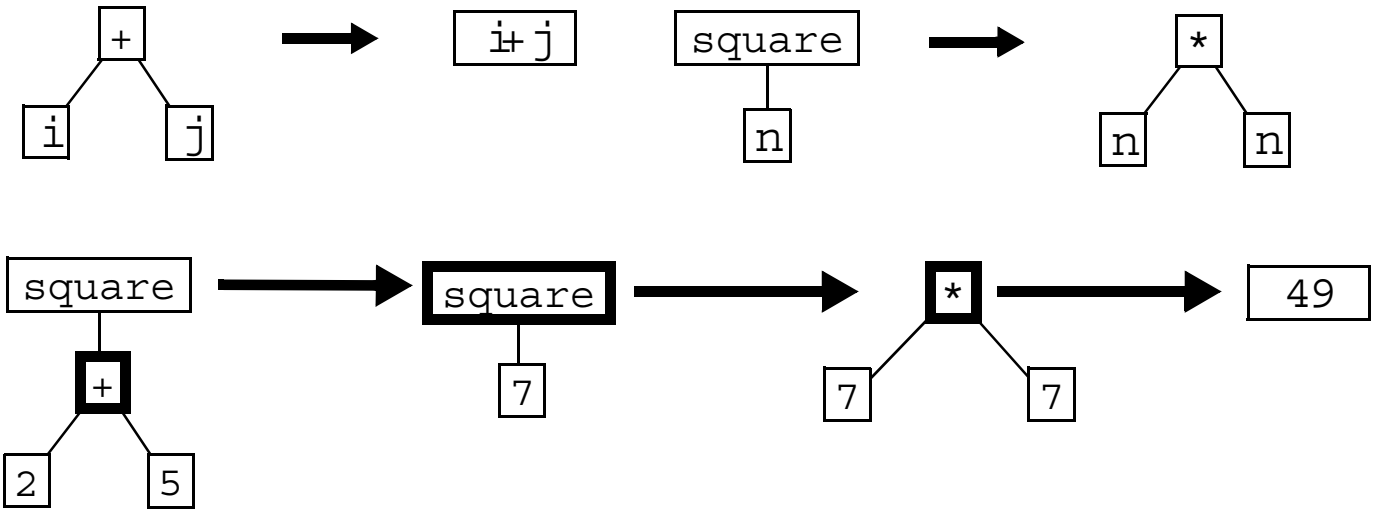
berechnen alle Normalformen

strikte Auswertung

eager, applicative, call-by-value, leftmost-innermost

- erst wird das Argument ausgewertet (`p+q`),
- dann wird die Funktion (`square`) aufgerufen

```
square (2 + 5) = square (7)
               = 7 * 7
               = 49
```

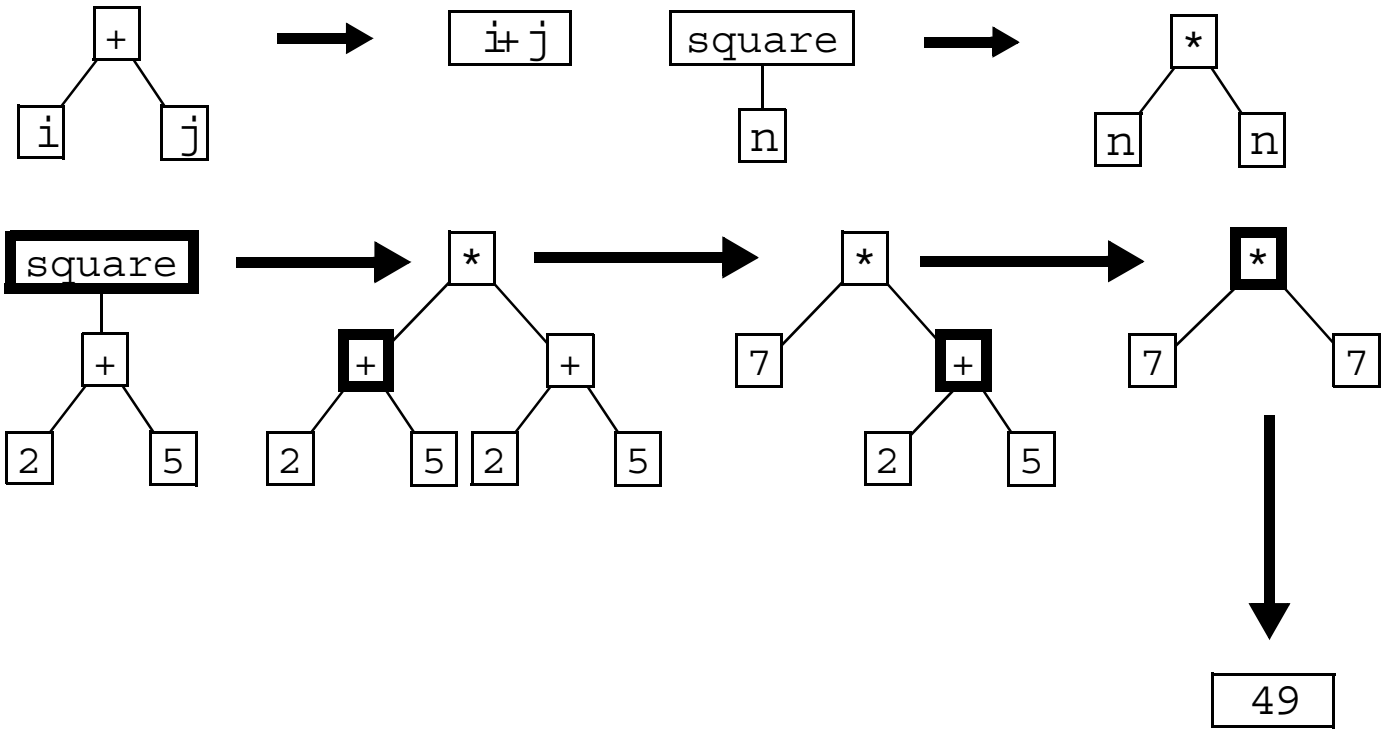


normalisierende Auswertung

normalorder, call-by-name, leftmost-outermost

- das Argument (`p+q`) wird unausgewertet übergeben
- in der Funktion (`square`) wird es bei Bedarf ausgewertet
  (* und + funktionieren aber nur für ausgewertete Zahlen)

```
square (2 + 5) = (2 + 5) * (2 + 5)
               = 7 * (2 + 5)
               = 7 * 7
               = 49
```

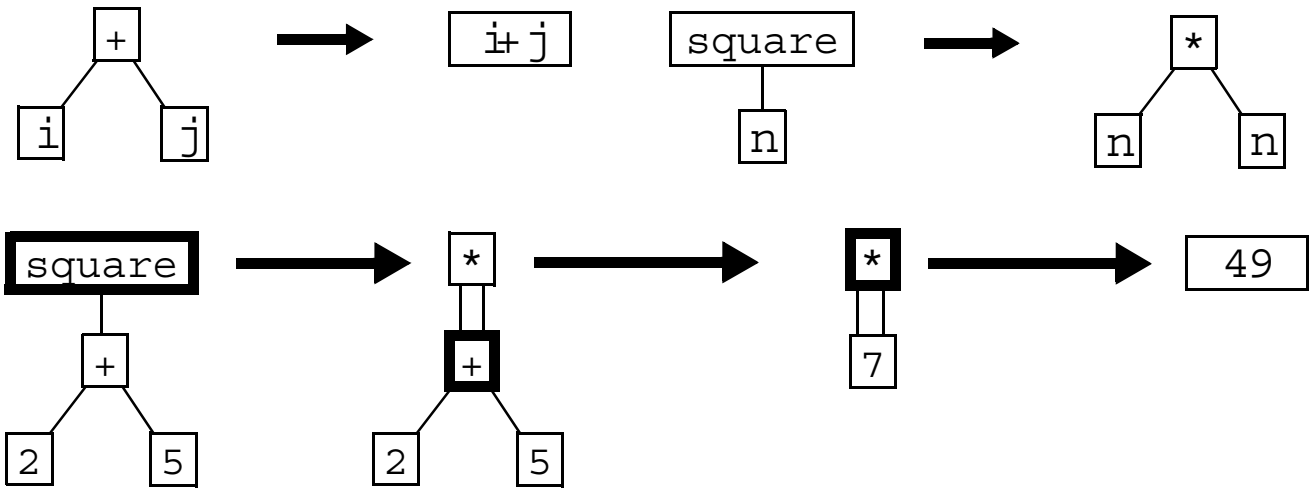


verzögerte Auswertung (lazy evaluation)

Eine Verbesserung der normalisierenden Auswertung

- das Argument an der ersten Stelle auswerten,
  - an der die Funktion es braucht
  - an allen weiteren Stellen diesen Wert benutzen

```
square (2 + 5) = (2 + 5) * (2 + 5)
               = 7 * 7
               = 49
```



verzögerte Auswertung

- kann auch alle irgendwie auswertbaren Ausdrücke auswerten
- ist immer noch ineffizienter als strikte Auswertung

Unterschiede der Auswertungsstrategien

Beispiel (Haskell)

```
i / 0 = error
i / j = m where m ist der Quotient von i und j
ite true b b' = b
ite false b b' = b'
...
n = 0
```

- strikte Auswertung

```
ite (n>0) (1/n > 0.5) false
                    =  ite false (t/n > 0.5) false
                    =  ite false (error > 0.5) false
                    =  ite true error false
                    =  error
```

- normalisierende Auswertung (oder verzögerte)

```
ite (n>0) (1/n > 0.5) false
                    =  ite false (t/n > 0.5) false
                    =  false
```

Ursache: `ite` ist nicht strikt (im 2. und 3. Argument)

Unterschied

Wenn ein Ausdruck irgendwie ausgewertet werden kann,

dann auch normalisierend

aber

normalisierende Auswertung ist ineffizient

auch verzögerte Auswertung ist ineffizienter als strikte

Seiteneffekte und Auswertungsreihenfolge

normalisierende und verzögerte Auswertung

- vertragen sich nicht mit Seiteneffekten
- strikte Auswertung

```
square(read) = square (7)
             = square (7)
             = 7 * 7
             = 49
```

- normalisierende Auswertung

```
square(read) = square (read)
             = (read) * (read)
             = 7 * (read)
             = 7 * 3
             = 21
```

Problem

- man muß den Rumpf der Abstraktion kennen,
  - um den Wert des Parameters zu wissen
- das widerspricht dem Sinn der Abstraktion (trenne WAS von WIE)

Auswertungsstrategien in Programmiersprachen

Strategien sind meistens fest verdrahtet

strikte Auswertung
- ist die Norm für imperative Sprachen (aber Algol-60!)

normalisierende Auswertung
- gibt es nur in älteren Sprachen (Algol-60, Lisp)

verzögerte Auswertung
- hat sich für einige rein funktionale Sprachen durchgesetzt
  - (Miranda, Haskell)

Übungen: Seiteneffekte in Ada-Funktionen

Wie kann man Seiteneffekte ausschließen?

durch Beschränkungen des Funktionsrumpfes

Ziel

1. gleiche Funktionsaufrufe in einem Ausdruck liefern das Gleiche

```
x := F(a+b,3) + F(a+b,3)
```

2. gleiche Funktionsaufrufe liefern immer das Gleiche

```
x := F(a+b,3);
z := z+1;
y := F(a+b,3);
```

Stufe 1

alle Parameter sind **in**

alle globalen Größen bleiben unverändert

- keine Verwendung als Ziel einer Zuweisung

nur globale Funktionen werden aufgerufen

Stufe 1

alle Parameter sind **in**

nur konstante globale Größen werden verwendet

nur globale Funktionen werden aufgerufen

Übungen: Abstraktion in Ada

Welche Arten der Abstraktion gibt es?

- Funktionen (und Operationen)
- Prozeduren
- keine Selektoren

Gilt das Abstraktionsprinzip?

- jein

Wie werden Parameter übergeben?

- einfache Typen: **in-out**
- zusammengesetzte Typen: Referenzen oder **in-out**

Gilt das Korrespondenzprinzip?

```
prodecure P (F: in T) is
   begin …end P;

P(A);          declare
                  F: constant T := A
               begin …end;

prodecure P (F: out T) is
   begin …end P;

P(A);          declare
                  F: constant T
               begin …F  := A; end;

prodecure P (F: var in T) is
   begin …end P;

P(A);          declare
                  F: constant T renames A
               begin …end;
```

Übungen: Referenzparameterübergabe in Pascal

Betrachten Sie die Pascal-Prozedur

```
procedure multiply (var m, n : Integer);
   begin
   m := m * n;
   writeln(m,n)
   end;
```

Was drucken die Aufrufe (für `i=1` und `j=3`)

```
multiply (i, j);
multiply (i, i);
```

Was lieferten sie, wenn `m` und `n` **`in out`**-Parameter wären?

Übungen: "Namensparameter" in Pascal

Jensons device

```
var a: array [1..10] of Real;
function jd (name ai : Real) : Real;
      var sum: ReaL ; i : Integer;
   begin
   for i:=1 to 10 do sum:= sum + ai;
   jd := sum
   end;
write(jd(a[i]))
```

Was tut diese Funktion? Wie kann man das in Pascal erreichen?

```
function access (i : Integer) : Real;
   begin access := a[i] end;
function jd (f : procedure (Real): Real) : Real;
      var sum: Real ; i : Integer;
   begin
   for i:=1 to 10 do sum:= sum + f(i);
   jd := sum
   end;
write(jd(acc))
```

Übungen: Pascal und das Korrespondenzprinzip

Pascal erfüllt nicht einmal annähernd das Korrespondenzprinzip

- Es gibt keinen Parametermechanismus für die Konstantendefinition
- Es gibt keine Variablenumbenennung für die Variablenparameter
- Es gibt keine initialisierende Variablenvereinbarung für die Wertparameter

Übung: Erweitern Sie Pascal, so daß das Korrespondenzprinzip gilt