Programmiersprachen
2. Überleitung

von Konzepten

zu Paradigmen

Konzepte

grundlegende Konzepte
- Werte
- Speicher
- Bindung
- Abstraktion

fortgeschrittene Konzepte
- Kapselung
- Ablaufsteuerung
- Typsysteme

insgesamt also

eine *analytische* Sicht auf die *Bestandteile* von Programmiersprachen

Paradigmen

Programmierstile

Kombination von Konzepten zu *Programmierstilen*

typische Sprachen

etablierte Paradigmen

imperatives Programmieren (Ada)

objektorientiertes Programmieren (Java, Eiffel)

funktionales Programmieren (ML, Haskell)

logisches Programmieren (Prolog)

exotischere Paradigmen

nebenläufiges Programmieren

visuelles Programmieren

Paradigmen
1. Imperatives Programmieren

das lange dominierende Paradigma

aber nicht das älteste! (Lisp ist älter als Fortran!)

Inhalt

Was ist typisch imperativ?

Vor- und Nachteile und Alternativen

Fallstudie Ada

Das imperative Ausführungsmodell

die von-Neumann-Maschine
- Programmspeicher mit direktem Zugriff auf *einzelne* Befehle (über Adressen)
- Speicher mit direktem Zugriff auf *einzelne* Zellen (über Adressen)
- *sequentielle* Ausführung der Instruktionen
- *zellenweises* Überschreiben des Speichers

die Maschine wird schön verpackt
- zusammengesetzte Werte
- zusammengesetzte Variablen
- zusammengesetzte Befehle
- Prozeduren und Parameterisierung
- statische Typisierung

trotzdem bleibt es eine von-Neumann-Maschine

Beispiel: Funktionen

```
function power (b : Real; n : Integer) return Real is
      p : Real := 1.0;
  begin
  for i in range 1 .. abs (n) loop p := p * b; end loop
  if n >= 0 then return p else return 1.0 / p end
  end power;

function power (b : Real; n : Integer) return Real is
  begin
  if n = 0 then power := 1.0
  else if n > 0 then return b * power (b, n-1)
  else return 1.0 / power (b, -n)
  end
```

welche Fassung ist "imperativer"?

Beispiel: Zeiger

```
type Customer is
  record name: Name; address : Address; number : Number
  end record;
type NameTree;
type NameTree is accessrecord
                          keyname : Name;
                          details : access Customer;
                          left, right : NameTree
                        end record;
type NumberTree is …;  (* wie NameTree *)
var directory : record
                  byname   : NameTree;
                  bynumber : NumberTree;
                end record;
```


typisch imperativ

Maschinenähe

Maschinenunabhängigkeit

Sequenzialität

Speicherbewusstsein

Typisierung

Modularität

Plattformunabhängigkeit

Probleme

- *goto statement considered harmful*
- *global variables considered harmful*
- *pointers are the gotos of data structures*

die Programmiersprache Ada

Entwerfer

Jean Ichbiah et al., im Auftrag des *US Department of Defence*, 1979

1995: Erweiterung um objektorientierte Konzepte

Zweck

die *universelle* Sprache für *große* und *eingebettete* Systeme

Entwurfsziele

Wiederverwendbarkeit, Sicherheit, leichte Lesbarkeit

Zusammenfassung des *state of the art* bei imperativen Sprachen

Geschichte

Der Ausweg des *DoD* aus der Softwarekrise?

Vorgänger: Pascal

Datentypen in Ada

einfache Datentypen

vordefiniert: `boolean`, `integer`, `float`, `character`, ...

• verschiedene, abfragbare Genauigkeiten (es gab noch keinen IEEE-Standard)

neue *fixed* und *float*-Typen können definiert werden:

```
type Fraction is
  delta System.fine_delta range -1.0 .. 1.0;
```

zusammengesetzte Datentypen

`record` mit gesicherten Varianten

`array` mit dynamischen Grenzen und diskreten Indexbereichen

`access` (getypte, überprüfte Zeiger)

Besonderheit (für die Systemprogrammierung)

Repräsentationsklauseln erlauben genaue Festlegung des Speicherlayouts

Untertypen in Ada

konkrete Untertypen (Beschränkung der Wertemenge)

Teilbereiche und Genauigkeitseinschränkungen von Zahlen

```
subtype Natural is Integer range 0..maxint;
type Money is delta 0.01 digits 15;
subtype Salary is Money digits 10;
```

Indexbereiche bei Feldern

```
type String is array (Natural range <>) of Character
subtype ShortString is String (1..8);
```

Diskriminanten bei Verbunden

```
type Person (female: boolean) is
  record ... end record;
subtype Male is Person(false);
```


Typisierung in Ada

Typen

statische Typüberprüfung

namentliche Typäquivalenz

Untertypen

dynamische Typüberprüfung

strukturelle Typäquivalenz

Ausdrücke in Ada

Ausdrücke können beliebige Typen haben

Literale einfacher Typen

Aggregate zusammengesetzer Typen (**`array`**, **`record`**)

Variablen beliebigen Typs

Funktionsaufrufe beliebiger Typen

Operationsaufrufe sind spezielle Funktionsaufrufe

```
x + 3.0  ?  „+“ (x, 3.0)
```

typisch imperativ

• es gibt *keine* bedingten Ausdrücke (**`if`**, **`case`**) oder Blockausdrücke

Befehle in Ada

alle Befehle zum *strukturierten Programmieren*

Wertzuweisung

`if`

`case`

Schleifen in allgemeinerer Form

```
[<<M>>] [while C | for V in Range]
loop
... exit M when C
... exit M when C
end loop
```


Vereinbarungen in Ada

bindbare Größen

- Typen und Untertypen (**`type`**, **`subtype`**)
- *Objekte*, Variablen und Konstanten (*X* : [**`constant`**] *T* := *E*)
- Prozeduren und Funktionen (**`procedure`**, **`function`**)
  Moduln (**`package`**)
- Augaben und Einträge (**`task`**, **`entry`**)
- Ausnahmen (**`exception`**)

Definition *versus* Einführung

	`subtype`	*versus*	**`type`**
X : *T*	**`renames`** *Y*	*versus*	*X* : *T* := *E*
`procedure` *P...*	**`renames`** *Q*	*versus*	**`procedure`** *P...* **`is`** *C*
`package` *P...*	**`renames`** *Q*	*versus*	**`package`** *P...* **`is`** *C*
E : **`exception`**	**`renames`** *F*	*versus*	*E* : **`exception`**

Blöcke in Ada

Blockbefehl

```
declare D
begin
  C
end
```

Paketrümpfe und Prozedurrümpfe sind Blockbefehle

Bindung

statisch

Sichtbarkeit

strikt linear

Abstraktion in Ada

Abstraktionen

Funktionen (beliebigen Resultattyps), Prozeduren und generische Pakete

- Abstraktionen sind Werte *dritter* Klasse
- sie können nur als generische Parameter zu übergeben werden!

„abstrakte" Parametermechanismen *in* / *out* / *in-out*

- *kopierender* Mechanismus für *einfache* Typen
- *unspezifiziert* für *zusammengesetzte* Typen
- *definitorischer* Mechanismus für *Objekte* usw. (Ada 95)

syntaktischer Schnickschnack

- namentliche oder positionelle Assoziation von aktuellen an formale Parameter
- *default*-Parameterwerte für fehlende aktuelle Parameter

Kapselung und Typisierung in Ada

Kapselung (`package`)

- syntaktische Trennung von Schnittstelle (*was?*) und Rumpf (*wie?*)
- Privatvereinbarungen in der Schnittstelle für die getrennte Übersetzung

geeignet für

- abstrakte Datentypen
- Objekte und Objektklassen (in Ada 80 *ohne* Vererbung!)
- parametrisierte Pakete (`generic`) mit Typparametern

das Typsystem von Ada

Überladen

kontextabhängig

für Funktionen und Prozeduren, auch für parameterlose Funktionen

parametrische Polymorphie

so gut wie nicht (`null` :: `access` ?)

systematische imperative Untertypen

Teilbereiche, Genauigkeiten, Felder, Verbunde

keine Typerweiterungen (in Ada 80)

Ablaufsteuerung in Ada

Sprünge

`goto`

Ausgänge

`exit`

`return`

Ausnahmen

`exception`

wie in der Vorlesung behandelt

Nebenläufigkeit

Form

Aufgaben (`task`)

Einträge (`entry`)

Kommunikation

Rendezvous

siehe auch später?

Zusammenfassung: Ada 80

ziemlich prinzipientreu

Typvollständigkeit	*ja*
Qualifikation:	*ja*
Abstraktion:	*nein* (nicht für Variablen)
Korrespondenz	*ja*

Mängel

- keine Prozeduren als Parameter (nur mit generischen Funktionen)
- `private` Vereinbarungen
- syntaktischer Reichtum stiftet leicht Verwirrung

Typerweiterung in Ada-95

erweiterbare Verbunde

```
type Point is tagged record x, y: Float; end re-
cord;
type Circle is new Point with
                record r: Float; end record;
O: Point := (1.0,1.0); C: Circle := (0.0,0.0, 3.0);
O := Point(C); C := O with 6.0;
```

Vererbung in Ada 95

vererbbare Prozeduren

```
function distance (p,q : Point) return Float is
  begin return sqrt(sqr(p.x-q.x) + sqr(p.y-q.y)) end;
procedure move (in out p: Point, dx, dy: Float) is
  begin p.x := p.x+dx, p.y := p.y+dy) end;
function area (p: Point'Class) return Float is
  begin return 0.0 end;
```

Überschreiben von Funktionen für erweiterte Typen

```
function area (c: Circle) return Float is
  begin return pi*sqr(p.r) end;
```

dynamic dispatch

```
fig: Point'Class;
fig:= ...; ... area(fig)...
```


Beispiel quicksort

```
generic
  typeItem is private;
  typeItemSequence is array (Integer range <>) of Item;
with function precedes (x, y : Item) return Boolean;
procedure sort (items : in out ItemSequence);
procedure sort (items : in out ItemSequence) is
  left, right : Integer;
begin
  if items'first < items'last then
    partition (items, left, right);
    sort (items(items'first..right));
    sort (items(left..items'last));
  end if;
end sort;
```


Beispiel partition

```
procedure partition (items : in out ItemSequence;
                     left, right : out Integer) is
  l : Integer := items'first; r : Integer := items'last;
  pivot : constant Item := items(l); t: Item;
begin
  loop
    while precedes(items(l), pivot) loop l:=l+1; end loop;
    while precedes(pivot, items(r)) loop r:=r-1; end loop;
    exit when l > r;
    t := items(l); items(l) := items(r); items(r) := t);
    l := l+1;  r := r-1;
    exit when l > r;
  end loop;
  left := l;  right := r;
end partition;
```


Beobachtungen

Abstraktion

das Paket ist gut wiederverwendbar

Effizienz

das Paket nutzt Speicher und Zeit effektiv aus

Verständlichkeit

Zustandsveränderungen sind nicht leicht nachvollziehbar

Effizienz hat seinen Preis!

Programmiersprachen
2. Überleitung

von Konzepten

zu Paradigmen

Konzepte

grundlegende Konzepte
- Werte
- Speicher
- Bindung
- Abstraktion

fortgeschrittene Konzepte
- Kapselung
- Ablaufsteuerung
- Typsysteme

insgesamt also

eine *analytische* Sicht auf die *Bestandteile* von Programmiersprachen

Paradigmen

Programmierstile

Kombination von Konzepten zu *Programmierstilen*

typische Sprachen

etablierte Paradigmen

imperatives Programmieren (Ada)

objektorientiertes Programmieren (Java, Eiffel)

funktionales Programmieren (ML, Haskell)

logisches Programmieren (Prolog)

exotischere Paradigmen

nebenläufiges Programmieren

visuelles Programmieren

Paradigmen
1. Imperatives Programmieren

das lange dominierende Paradigma

aber nicht das älteste! (Lisp ist älter als Fortran!)

Inhalt

Was ist typisch imperativ?

Vor- und Nachteile und Alternativen

Fallstudie Ada

Das imperative Ausführungsmodell

die von-Neumann-Maschine
- Programmspeicher mit direktem Zugriff auf *einzelne* Befehle (über Adressen)
- Speicher mit direktem Zugriff auf *einzelne* Zellen (über Adressen)
- *sequentielle* Ausführung der Instruktionen
- *zellenweises* Überschreiben des Speichers

die Maschine wird schön verpackt
- zusammengesetzte Werte
- zusammengesetzte Variablen
- zusammengesetzte Befehle
- Prozeduren und Parameterisierung
- statische Typisierung

trotzdem bleibt es eine von-Neumann-Maschine

Beispiel: Funktionen

```
function power (b : Real; n : Integer) return Real is
      p : Real := 1.0;
  begin
  for i in range 1 .. abs (n) loop p := p * b; end loop
  if n >= 0 then return p else return 1.0 / p end
  end power;

function power (b : Real; n : Integer) return Real is
  begin
  if n = 0 then power := 1.0
  else if n > 0 then return b * power (b, n-1)
  else return 1.0 / power (b, -n)
  end
```

welche Fassung ist "imperativer"?

Beispiel: Zeiger

```
type Customer is
  record name: Name; address : Address; number : Number
  end record;
type NameTree;
type NameTree is accessrecord
                          keyname : Name;
                          details : access Customer;
                          left, right : NameTree
                        end record;
type NumberTree is …;  (* wie NameTree *)
var directory : record
                  byname   : NameTree;
                  bynumber : NumberTree;
                end record;
```


typisch imperativ

Maschinenähe

Maschinenunabhängigkeit

Sequenzialität

Speicherbewusstsein

Typisierung

Modularität

Plattformunabhängigkeit

Probleme

- *goto statement considered harmful*
- *global variables considered harmful*
- *pointers are the gotos of data structures*

die Programmiersprache Ada

Entwerfer

Jean Ichbiah et al., im Auftrag des *US Department of Defence*, 1979

1995: Erweiterung um objektorientierte Konzepte

Zweck

die *universelle* Sprache für *große* und *eingebettete* Systeme

Entwurfsziele

Wiederverwendbarkeit, Sicherheit, leichte Lesbarkeit

Zusammenfassung des *state of the art* bei imperativen Sprachen

Geschichte

Der Ausweg des *DoD* aus der Softwarekrise?

Vorgänger: Pascal

Datentypen in Ada

einfache Datentypen

vordefiniert: `boolean`, `integer`, `float`, `character`, ...

• verschiedene, abfragbare Genauigkeiten (es gab noch keinen IEEE-Standard)

neue *fixed* und *float*-Typen können definiert werden:

```
type Fraction is
  delta System.fine_delta range -1.0 .. 1.0;
```

zusammengesetzte Datentypen

`record` mit gesicherten Varianten

`array` mit dynamischen Grenzen und diskreten Indexbereichen

`access` (getypte, überprüfte Zeiger)

Besonderheit (für die Systemprogrammierung)

Repräsentationsklauseln erlauben genaue Festlegung des Speicherlayouts

Untertypen in Ada

konkrete Untertypen (Beschränkung der Wertemenge)

Teilbereiche und Genauigkeitseinschränkungen von Zahlen

```
subtype Natural is Integer range 0..maxint;
type Money is delta 0.01 digits 15;
subtype Salary is Money digits 10;
```

Indexbereiche bei Feldern

```
type String is array (Natural range <>) of Character
subtype ShortString is String (1..8);
```

Diskriminanten bei Verbunden

```
type Person (female: boolean) is
  record ... end record;
subtype Male is Person(false);
```


Typisierung in Ada

Typen

statische Typüberprüfung

namentliche Typäquivalenz

Untertypen

dynamische Typüberprüfung

strukturelle Typäquivalenz

Ausdrücke in Ada

Ausdrücke können beliebige Typen haben

Literale einfacher Typen

Aggregate zusammengesetzer Typen (**`array`**, **`record`**)

Variablen beliebigen Typs

Funktionsaufrufe beliebiger Typen

Operationsaufrufe sind spezielle Funktionsaufrufe

```
x + 3.0  ?  „+“ (x, 3.0)
```

typisch imperativ

• es gibt *keine* bedingten Ausdrücke (**`if`**, **`case`**) oder Blockausdrücke

Befehle in Ada

alle Befehle zum *strukturierten Programmieren*

Wertzuweisung

`if`

`case`

Schleifen in allgemeinerer Form

```
[<<M>>] [while C | for V in Range]
loop
... exit M when C
... exit M when C
end loop
```


Vereinbarungen in Ada

bindbare Größen

- Typen und Untertypen (**`type`**, **`subtype`**)
- *Objekte*, Variablen und Konstanten (*X* : [**`constant`**] *T* := *E*)
- Prozeduren und Funktionen (**`procedure`**, **`function`**)
  Moduln (**`package`**)
- Augaben und Einträge (**`task`**, **`entry`**)
- Ausnahmen (**`exception`**)

Definition *versus* Einführung

	`subtype`	*versus*	**`type`**
X : *T*	**`renames`** *Y*	*versus*	*X* : *T* := *E*
`procedure` *P...*	**`renames`** *Q*	*versus*	**`procedure`** *P...* **`is`** *C*
`package` *P...*	**`renames`** *Q*	*versus*	**`package`** *P...* **`is`** *C*
E : **`exception`**	**`renames`** *F*	*versus*	*E* : **`exception`**

Blöcke in Ada

Blockbefehl

```
declare D
begin
  C
end
```

Paketrümpfe und Prozedurrümpfe sind Blockbefehle

Bindung

statisch

Sichtbarkeit

strikt linear

Abstraktion in Ada

Abstraktionen

Funktionen (beliebigen Resultattyps), Prozeduren und generische Pakete

- Abstraktionen sind Werte *dritter* Klasse
- sie können nur als generische Parameter zu übergeben werden!

„abstrakte" Parametermechanismen *in* / *out* / *in-out*

- *kopierender* Mechanismus für *einfache* Typen
- *unspezifiziert* für *zusammengesetzte* Typen
- *definitorischer* Mechanismus für *Objekte* usw. (Ada 95)

syntaktischer Schnickschnack

- namentliche oder positionelle Assoziation von aktuellen an formale Parameter
- *default*-Parameterwerte für fehlende aktuelle Parameter

Kapselung und Typisierung in Ada

Kapselung (`package`)

- syntaktische Trennung von Schnittstelle (*was?*) und Rumpf (*wie?*)
- Privatvereinbarungen in der Schnittstelle für die getrennte Übersetzung

geeignet für

- abstrakte Datentypen
- Objekte und Objektklassen (in Ada 80 *ohne* Vererbung!)
- parametrisierte Pakete (`generic`) mit Typparametern

das Typsystem von Ada

Überladen

kontextabhängig

für Funktionen und Prozeduren, auch für parameterlose Funktionen

parametrische Polymorphie

so gut wie nicht (`null` :: `access` ?)

systematische imperative Untertypen

Teilbereiche, Genauigkeiten, Felder, Verbunde

keine Typerweiterungen (in Ada 80)

Ablaufsteuerung in Ada

Sprünge

`goto`

Ausgänge

`exit`

`return`

Ausnahmen

`exception`

wie in der Vorlesung behandelt

Nebenläufigkeit

Form

Aufgaben (`task`)

Einträge (`entry`)

Kommunikation

Rendezvous

siehe auch später?

Zusammenfassung: Ada 80

ziemlich prinzipientreu

Typvollständigkeit	*ja*
Qualifikation:	*ja*
Abstraktion:	*nein* (nicht für Variablen)
Korrespondenz	*ja*

Mängel

- keine Prozeduren als Parameter (nur mit generischen Funktionen)
- `private` Vereinbarungen
- syntaktischer Reichtum stiftet leicht Verwirrung

Typerweiterung in Ada-95

erweiterbare Verbunde

```
type Point is tagged record x, y: Float; end re-
cord;
type Circle is new Point with
                record r: Float; end record;
O: Point := (1.0,1.0); C: Circle := (0.0,0.0, 3.0);
O := Point(C); C := O with 6.0;
```

Vererbung in Ada 95

vererbbare Prozeduren

```
function distance (p,q : Point) return Float is
  begin return sqrt(sqr(p.x-q.x) + sqr(p.y-q.y)) end;
procedure move (in out p: Point, dx, dy: Float) is
  begin p.x := p.x+dx, p.y := p.y+dy) end;
function area (p: Point'Class) return Float is
  begin return 0.0 end;
```

Überschreiben von Funktionen für erweiterte Typen

```
function area (c: Circle) return Float is
  begin return pi*sqr(p.r) end;
```

dynamic dispatch

```
fig: Point'Class;
fig:= ...; ... area(fig)...
```


Beispiel quicksort

```
generic
  typeItem is private;
  typeItemSequence is array (Integer range <>) of Item;
with function precedes (x, y : Item) return Boolean;
procedure sort (items : in out ItemSequence);
procedure sort (items : in out ItemSequence) is
  left, right : Integer;
begin
  if items'first < items'last then
    partition (items, left, right);
    sort (items(items'first..right));
    sort (items(left..items'last));
  end if;
end sort;
```


Beispiel partition

```
procedure partition (items : in out ItemSequence;
                     left, right : out Integer) is
  l : Integer := items'first; r : Integer := items'last;
  pivot : constant Item := items(l); t: Item;
begin
  loop
    while precedes(items(l), pivot) loop l:=l+1; end loop;
    while precedes(pivot, items(r)) loop r:=r-1; end loop;
    exit when l > r;
    t := items(l); items(l) := items(r); items(r) := t);
    l := l+1;  r := r-1;
    exit when l > r;
  end loop;
  left := l;  right := r;
end partition;
```


Beobachtungen

Abstraktion

das Paket ist gut wiederverwendbar

Effizienz

das Paket nutzt Speicher und Zeit effektiv aus

Verständlichkeit

Zustandsveränderungen sind nicht leicht nachvollziehbar

Effizienz hat seinen Preis!