

Einführung in die Programmiertechnik

Konstruktion neuer Datentypen

Datentypen

- Ein *Datentyp* ist eine Menge von Daten zusammen mit einer Familie von Operationen
 - abstrakter Datentyp: beschrieben wird lediglich die Menge und die Semantik der Operationen, nicht aber die interne Repräsentation der Daten oder die Implementierung der Operationen

Mengenkonstruktionen

- Konstruktion neuer Mengen (evtl. auf Basis existierender Mengen M, M_i)
- Aufzählung: Menge $\{e_1, e_2, e_3, \dots, e_n\}$
 - C, C++, Java 5: enum (*enumerations*)
- Teilmengen: Gegeben Prädikat $P(x)$, bilde $\{x \in M \mid P(x)\}$
- k -te Potenz: Gegeben $k \in \mathbb{N}$: Menge M^k aller k -Tupel (a_1, a_2, \dots, a_k) mit $a_i \in M$
 - C, C++: Array (Felder) “von M ” der Länge k
 - Python: Liste (Beschränkung auf Länge per Konvention)
- Kartesisches Produkt: $M_1 \times M_2 \times \dots \times M_k$: Menge der k -Tupel (a_1, a_2, \dots, a_k) mit $a_i \in M_i$
 - C: struct
 - Python: Tupel, Klasse

Mengenkonstruktionen (2)

- Disjunkte Vereinigung: Gegeben M_i : $M = \{(i, m) \mid i \in I \wedge m \in M_i\}$
 - C: union (aber: keine *discriminated union*)
 - Python: Konvention in Variablenverwendung
- Potenzmenge: $\wp(M)$ (Menge aller Teilmengen von M)
 - Python: set
- Folgenraum: M^* (Menge aller endlichen Folgen aus Elementen von M)
 - Python: Liste
- Unendliche Folgen: M^∞ (Menge aller unendlichen Folgen von Elementen aus M)
 - beschränkt auf berechenbare Folgen
 - funktionale Sprachen, Python: Generatoren

Mengenkonstruktionen (3)

- Funktionenraum: Menge aller Abbildungen von I nach M
 - Funktionen
 - Python, C++, Java: Dictionaries (Mappings)
- Induktive Definition: Definition der Menge durch ein Induktionsschema

Listen

- nummerierte Folgen von Werten
 - Indizierung beginnt bei 0
- Konvention: Werte sollten alle den gleichen Typ haben
 - Repräsentation von M^k und M^*
- Werte kommagetrennt in eckigen Klammern
 - [1,2,5]
 - ["Hallo", "Welt"]
 - []
- Operationen: $L+L$, L^*i , $L[i]$, $L[i:j]$
 - Beispiel: \mathbb{N}^k : $[0]^*k$
- Funktionen: `len()`
- Methoden: `.append(V)`, `.index(V)`, `.reverse()`, `.sort()`

Tupel

- Kreuzprodukt von k Typen
 - per Konvention: Im gleichen Kontext haben Tupel sollten Tupel immer die gleiche Länge haben, und die jeweiligen Elemente den gleichen Typ
- nicht änderbar (*immutable*): Elemente werden bei Erzeugung festgelegt
- Werte kommagetrennt
 - “Erika”, “Mustermann”, 42
 - Klammern um leeres Tupel: ()
 - Klammern um Tupel bei Funktionsaufruf: f((3, 4, -9))
- Operationen: $T+T$, $T*i$, $T[i]$, $T[i:j]$
 - zusätzlich: *tuple unpacking* bei Zuweisung
 - $x,y,z = t$
- Funktionen: $\text{len}(T)$

Kartesisches Produkt mittels Klassen

- Klassen: Zusammenfassung von Zustand (Daten) und Verhalten
 - Verzicht auf Verhalten: “Datensätze” (records, structures)
- Elemente des Datensatzes über Attributnamen ansprechbar

class Person:

```
def __init__(self, vorname, name, alter):
```

```
    self.vorname = vorname
```

```
    self.name = name
```

```
    self.alter = alter
```

```
p = Person("Erika", "Mustermann", 42)
```

```
print p.vorname
```

```
p.alter += 1
```

```
print p.alter
```


Dictionaries

- Mapping-Typ: Abbildung von Typ K auf Typ V
 - K: “Schlüssel” (key)
 - V: “Wert” (value)
 - “endliche Funktion”: Abbildung ist nur für endliche Teilmenge von K definiert
 - “Variable Funktion”: Abbildung kann geändert werden
 - nur bestimmte Typen als Schlüsseltyp erlaubt
 - i.d.R. “immutable” (Schlüssel verändert seinen Wert nicht)
- Notation: { k1:v1, k2:v2 }
- Operationen: D[k], k in D (ab Python 2.3)
- Funktionen: len(D)
- Methoden: .keys, .values, .items, .has_key

Ausnahmen für Containertypen

- **IndexError**: Index ist für Liste/Tupel ungültig
 - größer `len(T)` oder kleiner `-len(T)`
- **KeyError**: Schlüssel ist für Dictionary ungültig
 - bisher keine Zuweisung unter diesem Schlüssel

Potenzmengen

- Konstruktor `set()`
- Konstruktion aus Liste oder Tupel
- Operatoren: $e \in S$
- Methoden: `.add(e)`, `.remove(e)`, `.union(s2)`, `.intersection(s2)`

Dateien

- *Dateien* sind endliche Folgen, *Ströme (streams)* potentiell unendliche Folgen von Datensätzen eines beliebigen, aber fest gewählten Typs
 - Pascal: Festlegung des Typs mit Sprachmitteln
 - C, Java, Python, ...: Typ ist (fast immer) Byte
 - evtl. Folge von Zeichen, evtl. Folge von Zeilen
- **Dateien: Speicherung von Daten zwischen Programmläufen**
 - “persistenter” Speicher (*persistent memory*)
- **Ströme: Austausch von Daten zwischen aktiven Programmen**
 - z.B. TCP/IP-Verbindungen

Dateien (2)

- Lesemodus oder Schreibmodus
 - “append” (anhängen): Spezialform des Schreibmodus
- Binär- oder Textmodus
 - Unterschied (in Python 2) nur für Windows relevant:
 - Textmodus: Zeilenenden werden automatisch in CRLF konvertiert
 - Zeichen 26 (`\x1a`) kennzeichnet das Dateiende
- Dateizeiger (*file pointer*): Angabe der Lese- oder Schreibposition in der Datei
 - initial: Anfang der Datei (außer “append”)
- Spezialdateien für Terminalinteraktion: Standardeingabe, Standardausgabe, Standardfehlerausgabe

Dateien (3)

- Zyklus für Verarbeitung von Dateien:
 - Öffnen der Datei (Angabe des Dateinamens und des Modus)
 - Ein/Ausgabeoperationen (lesen, schreiben)
 - Schließen der Datei
- Python: Öffnen der Datei mit Funktion `open()`
 - `open(dateiname[, modus[, buffering]])`
 - mögliche Modi: 'r', 'w', 'a', ... (binär: 'rb', 'wb',...)
 - Modus ist optional: Standardmodus ist 'r'
 - buffering ist optional (Größe des internen Puffers); sollte i.d.R. weggelassen werden
 - Ergebnis von `open`: Datentyp `file`

Dateien (4)

- Operationen von file: `.read()`, `.read(laenge)`, `.readline()`, `.readlines()`, `.write(bytes)`, `.writelines(liste_von_bytes)`
- Schließen der Datei mit `.close()`
 - gepufferte Daten werden an das Betriebssystem übergeben
 - passiert u.U. auch automatisch, spätestens mit Ende des Programms

Induktiv definierte Typen

- Induktionsschema: Elemente des Typs werden induktiv aus anderen Elementen erzeugt
 - natürliche Zahlen
 - Listen
 - Bäume
 - Stacks
 - Strings
 - ...
- Implementierungsstrategie: Speicherung der Werte auf Halde (*heap*)
 - Speicherverwaltung: Anforderung und Freigabe von Speicher
 - Teilweise automatisch: Java (*garbage collection*), Python (Referenzzählung)

Induktive Definition: Ein Beispiel

- Dateiverzeichnis: verschachtelte Verzeichnisse + Dateien
- Def.: Ein **VBaum** besteht aus
 - einem **Dateinamen**, oder
 - einem Paar (**Verzeichnisname**, **VBaumListe**)
- Def.: Eine **VBaumListe** ist
 - leer, oder
 - besteht aus einem **VBaum** und einer **VBaumListe**
- Darstellung im Speicher: “besteht aus” kann nicht durch Zusammenfügen von Speicherblöcken entstehen
 - Größe eines Werts von VBaumListe ließe sich nicht angeben
- Lösung: Zeiger (*pointer*) / Referenzen
 - Variablen enthalten nicht den eigentlichen Wert, sondern nur einen Verweis (Adresse) des Werts

Schritt 1

- Induktiv-definierte Liste: Leer, oder Element + Liste
 - Ziel: explizite Definition, anstelle von Rückgriff auf Python-Listentyp
- Python: Alle Variablen sind Referenzvariablen
 - Leere Liste repräsentiert durch Spezialwert None
 - Nicht-leere Liste durch Datensatz von zwei Elementen: Baum, Restliste

class VBaumListe:

```
def __init__(self, baum, restliste):  
    self.baum = baum  
    self.restliste = restliste
```

- Verwendung:

```
liste = None  
liste = VBaumListe("foo.txt", liste)
```

Schritt 2

- Definition des Typs VBaum: Dateiname, oder Datensatz (Verzeichnisname, VBaumListe)
 - Dateiname und Verzeichnisname repräsentiert durch String-Objekte

class VBaum:

```
def __init__(self, verzeichnis, inhalt):  
    self.verzeichnis = verzeichnis  
    self.inhalt = inhalt
```

- Verwendung:

```
liste = None
```

```
liste = VBaumListe("foo.txt", liste)
```

```
liste = VBaumListe("bar.txt", liste)
```

```
baum = VBaum("foobar", liste)
```

```
liste = VBaumListe(baum, None)
```

```
baum = VBaum("baz", liste)
```

Funktionen als Werte

- Definition von Funktionen:
 - def name(parameter):
 - inhalt
- Aufruf von Funktionen: name(argumente)
- Verwendung von Funktionen als Werte: Zuweisung an Variablen, Einfügen in andere Typen, Funktionen als Ergebnis anderer Funktion (Funktionen höherer Ordnung)

```
name2 = name
```

```
fliste = [math.sin, math.cos]
```

```
fliste[0](3.14)
```

```
def mkinc(n):
```

```
    def inc(a):
```

```
        return a+n
```

```
    return inc
```

```
f = mkinc(6)
```

```
f(7)
```