Konzepte von Programmiersprachen
2. Speicher

Inhalt

Speicher
Variablen
Referenzen
Lebensdauer
Befehle

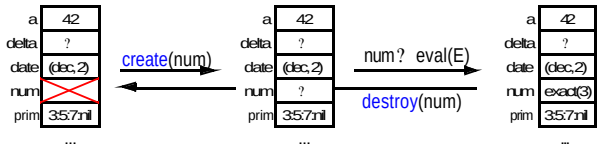
Speicher

Welche abstrakten Eigenschaften hat Speicher?

Speicher ist eine *Menge von benannten Speichezellen*

$S : X \to W \cup \{\text{undefiniert}\}$

Notiz: Anmerkungen zu Speicher



jede *Zelle* (x, S[x]) hat
- einen *Namen* $x \in X$ (eine *Variable*)
- einen *Status*: frei (wenn S[x] = $\bot$) oder belegt (sonst)
- einen *Inhalt*: irgendeinen Wert S[x] = $w \in W$ oder $\in$ *undefiniert*)

Frage: Ist dies realistisch?

Nein. Zellen haben eine feste Größe.

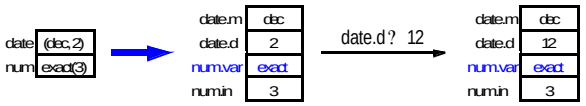
Datentypen

TT	::=	$t_1 = T_1; \ldots t_n = T_n;$	Datentypdefinitionen
T	::=	`Bool` | `Int` | `Char` | `Float` | ...	einfache Typen
	|	$t_1 \mid \ldots \mid t_n$	benutzerdefinierte Typnamen
	|	$T_1 \times \ldots \times T_n$	Produkt
	|	$T_1 + \ldots + T_n$	Summe
	|	$[u \,..\, o] \to T_2$	Feld (Funktionen später)

Welche Werte sind *speicherbar*?

nur einfache Werte passen in eine Speicherzelle
zusammengesetzte Werte werden auf mehrere Zellen verteilt

Speicherung direkt konstruierter Werte



Anmerkungen
- es gibt unsichtbare Komponentenvariablen
- Komponenten können selektiv überschrieben werden (manchmal: *nur so*!)
- Felder kommen später

Frage
- Können rekursive konstruierte Werte so gespeichert werden?
- Weshalb nicht?
- Wie dann?

Speicherstrukturen

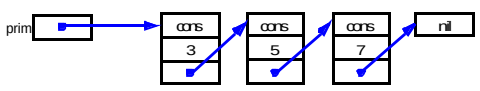
Variablen werden zu *speicherbaren Werten* gemacht

T	::=	`Bool` | `Int` | `Char` | `Float` ...
	|	$t_1 \mid \ldots \mid t_n$
	|	$T_1 \times \ldots \times T_n$
	|	$T_1 + \ldots + T_n$
	|	$T_1 \to T_2$
	|	$[u \,..\, o]\, T$
	|	$\uparrow T$ Verweise

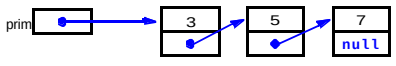
Notiz: Speicherstrukturen

- tatsächlich rekursive Typdefinitionen $t_i = T_i$ werden durch $t_i = \uparrow T_i$ ersetzt
- das geschieht in *allen* Sprachen
  in objektorientierten und funktionalen Sprachen wird dies nur *verborgen*!

Beispiel List = ? (Unit ? Int ? List)



Noch imperativer (nil = `null` **)** List = ? (Int ? List)



Variablen als Werte

List = ? (Int ? List)

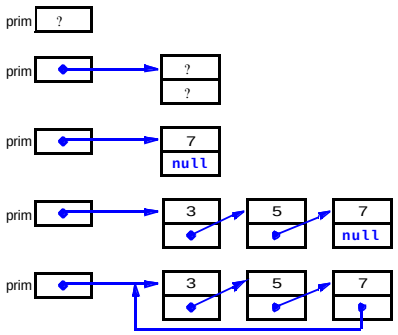
prim: List

`new`(prim)
prim↑.hd = ??
prim↑.tl = `null`

prim↑ = List(3, List(5, List(7,`null`)))

prim↑.tl↑.tl↑.tl = prim↑

Notiz: Variablen als Werte



Notiz: Variablen als Werte

- Variablen vom Typ ? *T* werden Zeiger genannt
- Ihr Inhalt ist eine Variable vom Typ *T* (eine Referenz) oder **`null`** oder ?
- Konstruktor: Allokation einer (*dynamischen*, *anonymen*) Variablen
  — mit dem Befehl **`new`**(*V*)
  — oder mit der Konstruktorfunktion *V* ? *T*(...)
- Selektor: Zugriff auf den Inhalt durch *V*? (partiell)

Mit Verweisen können rekursive Werte abgespeichert werden

aber auch Werte, die sich ohne Verweise nicht so bilden lassen

Zum Beispiel

- zyklische Listen
- Bäume mit „Verkettung“ von allen Blättern zur Wurzel
- oder beliebige „Spaghetti-Verkettungen“

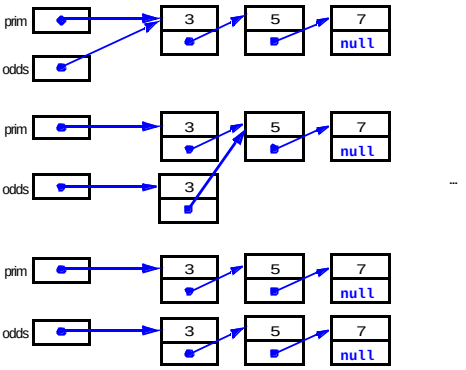
„Verweise sind die Sprünge der Datenstrukturen“

Wert- und Zeigersemantik

odds: List
odds ? ?prims

odds? .tl ? .tl ? .tl ? List(9, **`null`**)

Notiz: Wert- und Zeigersemantik



Notiz: Wert- und Zeigersemantik

Was wird ersetzt?

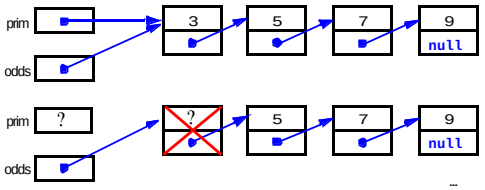
1. die in *odds* gespeicherte Variable ? *Zeigersemantik*
2. der Inhalt von *odds*? durch den Inhalt von *prim*?
3. auch (rekursiv) der Inhalt aller Variablen in *odds*? ? *Wertsemantik*

Abwägung

- Wertsemantik ist bei zusammengesetzten Werten *teuer*, gerade bei rekursiven (bei *zyklischen Verweisen* kann es richtig schwierig werden!)
- Zeigersemantik erzeugt Aliase: *odds* und *prim* verweisen auf *dieselben* Zellen
  Jedes spätere Überschreiben von *prim* ändert auch den Inhalt von *odds*.
- Lösung bei funktionalen Sprachen:
  Zeigersemantik, aber kein selektives Überschreiben
- Ändern heißt in einer Kopie ändern

Notiz: baumelnde Verweise

`dispose`(prims)



baumelnde Verweise

das Prinzip der Speichersicherheit

Variablen werden erst gelöscht, wenn es keinen Verweis mehr auf sie gibt

Wie können baumelnde Verweise verhindert werden?

`dispose` prüft, ob die Variable noch gezeigt wird

automatische Seicherbereinigung (*garbage collection*)

statische Feldvariablen

```
type Vector is array (1..9) of Real;
procedure readvector (out v : Vector); …;
…

a : Vector;
readvector(a);
```

dynamische Feldvariablen

```
type Vector is array (Integer range <>) of Real;
procedure readvector (out v : Vector); …;
…

a : Vector (1..9);
readvector(a);

…
read(m);
b : Vector (0..m);
readvector(b)
b := a;
```


Notiz: Feldvariablen

Arten von Variablen
- statisch: Feldgrenzen stehen fest
- dynamisch: Feldgrenzen stehen beim Anlegen der Variablen fest
- flexibel: Grenzen können sich bei jeder Operation anpassen

Konsequenzen für das Speichern von Feldvariablen?
- bei *dynamischen* und *flexiblen* Feldern müssen die *aktuellen Feldgrenzen* mit abgespeichert werden
- flexible Feldern sind Zeiger

Lebensdauer von Variablen

statisch (global oder lokal)
- von ihrer Vereinbarung
- bis dieVereinbarung ungültig wird

dynamisch
- vom Anlegen mit **new**
- bis zum Freigeben mit **dispose**
- *sicherer*: bis zur Freigabe durch Speicherbereinigung

persistent
- schon vor dem Starten des Programms
- auch nach Beenden des Programms

Notiz: Persistenz

welche persistenten Variablen gibt es?

Welche Wertemengen dürfen sie haben?

Weshalb haben transsiente und persistenrte Variablen verschieden Typen?

Forderung
- *jeder* Typ sollte ein- und ausgegeben werden können

Prinzip der Typvollständigkeit

Keine Operation soll in den Typen ihrer Operanden willkürlich eingeschränkt sein

Fragen

Welche Probleme macht das?
dynamische Typisierung

persistente Feldvariablen (Pascal)

```
program StatisticsAnalysis (stats);
type Country = (B,D,DK,E,F,GB,GR,I,IRL,L,NL,P);
     Statistics  = record
                     population: 0..100 000 000; ...
                   end;
var stats: array [Country] of Statistics;
procedure analyzestate;
var cy: Country;
     ...stats[cy].population ...
end;
begin
analytestate
end.
```


Befehle

C ::= V **:=** E
| **new**(V)
| p**(**E_1**,** … **,** E_k**)**
| **{** C_1**;** … **;** C_n **}**
| **if** E **then** C **else** C
| **select** E **case** l_1 : C_1 … **case** l_n : C_n **end**
| **while** E **do** C
| **repeat** C **until** E
| **for** V **in [** E_1, … , E_n **] do** C **end**

Semantik

die Ausführung eines Befehls verändert der Speicherzustand

Notiz: Befehle

1. Was haben diese Befehle gemeinsam?
2. Fehlen wichtige Befehle?

Anmerkungen

Prinzip

jeder Befehl hat einen Eintrittspunkt und einen Austrittspunkt

Kompositionalität *divide and conquer*

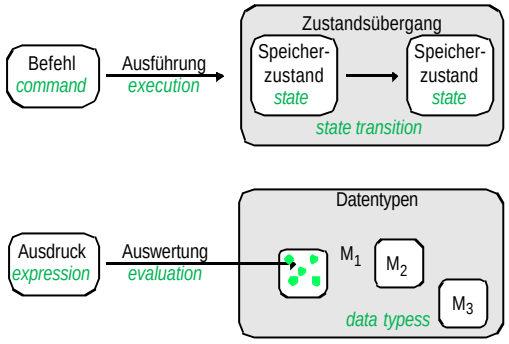
Verallgemeinerung(Ablaufsteuerung)

jeder Befehl kann einen Eintrittspunkt, aber *mehrere* Austrittspunkte haben

Datenstrukturen — Kontrollstrukturen

Produkt	—	Sequenz
Summe	—	Fallunterscheidung
Felder	—	Zählschleife
Sequenzen (Dateien)	—	Schleife
rekursive Typen (Listen)	—	*rekursive* Prozeduren

imperative *versus* applikative Façette



Ausdrücke mit Seiteneffekten

```
type Coeff is array Integer range <> of Real;
procedure horner(c: Coeff, x: Real) return Real is

  declare p: Real;
  begin p:= c(c'last);
  for i:= in reverse [0 .. n-1] do p:= p * x + c(i);
  return p

end horner;
```


ausdrucksorientierte Sprachen

Eigenschaft: *ausdrucksorientiert*

zwischen Befehlen und Ausdrücken wird nicht unterschieden

Konsequenz

- alle Befehle sind *Operationen* und liefern einen Wert (notfalls **`Unit`**)
- alle Ausdrücke können den Speicherzustand ändern

Fragen

1. Kennt Ihr ausdrucksorientierte Sprachen?
2. Vorteile?
3. Nachteile?