Einführung in die Informatik I (autip)

Dr. Stefan Lewandowski

Fakultät 5: Informatik, Elektrotechnik und Informationstechnik
Abteilung Formale Konzepte
Universität Stuttgart

7. November 2007

Inhalte heute:

- Funktionen
- Formatierte Ausgabe von integer mit Put
- Datentyp float
- Typkonvertierungen integer/float
- Formatierte Ausgabe von floats mit Put
- Arithmetische Ausdrücke

Letzte Woche ...

- Programm zur Berechnung der Fakultät
- unschön:
  - bisher keine Zuweisung der Art `x:=fak(n)` möglich
  - Ausgabe der Zahlen unschön
  - Standardeinstellungen sind für mehrzeilige Ausgaben gut geeignet (evtl. etwas breit), für einzelne Zahlen aber unnötig viele Leerzeichen

- Beispiel aus der Übung: Studienstiftung
- unschön:
  - Rundungsfehler auf ganze Euro $\rightsquigarrow$ Datentyp float
- wir sehen dabei gleich:
  - Strenges Typkonzept in Ada 95
  - Konvertierung zwischen Datentypen
  - Formatierte Ausgabe für Datentyp float

Zusammenfassung:

- ```
  function <Bezeichner> ( <Parameter-Liste> ) return <Datentyp> is
    <Deklarationen>
  begin
    <Anweisungs-Folge>
  end [<Bezeichner>];
  ```
- wenigstens eine `return`-Anweisung
- Datentyp `integer`:

  ```
  Put(<integer-Ausdruck> [, [width=>] <integer-Ausdruck>] );
  ```

  z.B. `Put(zahl,width=>5);`, `Put(a+b,width=>7);`, `Put(42,0);`
  - das ist nicht ganze Wahrheit, reicht aber für den Moment :-)
  ```

- Datentyp `float`:

  `Put(<float-A.> [, [fore=>] <integer-A.>] [, [aft=>] <integer-A.>]`

  z.B. `Put(zahl,exp=>0);`, `Put(a*b+c,aft=>2);`, `Put(x,3,4,0);`
- lässt man die Parameter-Bezeichner `fore`, `aft`, `exp` weg, so wird der erste Ausdruck für `fore` verwendet, der zweite für `aft` und der dritte für `exp`
- der Lesbarkeit halber Bezeichner mit hinschreiben

- Ada verfolgt ein strenges Typkonzept, d.h., es finden keine expliziten Typumwandlungen statt (Ausnahmen wie `natural`/`integer` – dies sind in Ada aber Unterbereiche und keine eigenständigen Typen – ggf. später)
- wo möglich können Typen explizit konvertiert werden: `<Datentyp>(<Ausdruck>)`, z.B. `Float(3*4)`, `Integer(Float(42)/5.7)`, ...
- Achtung!
  - Es können nicht beliebige Typumwandlungen durchgeführt werden (dies wird man in der Regel auch nicht wollen).
  - Es treten ggf. Rundungsfehler auf.

- Vorteile:
  - Durch das (strenge) Typkonzept werden bestimmte Flüchtigkeitsfehler vermieden.
  - Keine Mehrdeutigkeiten bei Ausdrücken wie z.B. `x:float; ...x*5;`
  - Tippfehler, die in andere Sprachen nicht erkannt werden, werden hier ggf. zu formalen Fehlern und können so vom Ausführer erkannt werden.
  - die Lesbarkeit und Wartbarkeit von Programmen wird erhöht.
- Die meisten Programmiersprachen unterstützen ein Typkonzept. Diese unterscheiden sich zum Teil aber erheblich. Ada setzt das Typkonzept sehr konsequent um. Andere Sprachen (z.B. C) sind hier deutlich laxer (betrachten Sie z.B. den „Typ“ boolean in der Sprache C).
- Ohne Typkonzept lassen sich Flüchtigkeitsfehler oft schwer finden.

Ein wenig Theorie - Zuweisungen und Ausdrücke:

- Die Zuweisung: auf der linken Seite einer Zuweisung „:=“ steht stets eine Variable, auf der rechten Seite stets ein Ausdruck, in EBNF ausgedrückt:
  `<Zuweisung> ::= <Bezeichner> := <Ausdruck> ;`
- wichtig ist dabei, dass der Ausdruck sich zu einen Wert vom selben Typ auswertet ($\rightsquigarrow$ starkes Typkonzept in Ada). Zwischenschritte bei der Auswertung des Ausdrucks können dabei durchaus von anderem Typ sein. Betrachten wir z.B. `Integer(Float(123/4)*5.67+0.8)` so würde zunächst `123/4` ganzzahlig dividiert werden (Ergebnis integer-Zahl 30), die 30 in float gewandelt werden, diese dann mit 5.67 multipliziert (Ergebnis float-Zahl 170.1), 0.8 addiert (float-Zahl 170.9) und dann bei der Typ-Umwandlung auf 171 gerundet. Dieser Wert würde dann ggf. einer integer-Variablen zugewiesen werden können, aber nicht einer float-Variablen.
- Ada 95 rundet korrekt (also Aufrunden ab .5) – dies ist bei vielen anderen Programmiersprachen anders!

Wichtig! Es wird immer der Wert eines Ausdrucks zugewiesen, nicht der Ausdruck selbst – Variablen sind Wertebehälter, keine Ausdrucksbehälter.
Ein Ausdruck ist eine Anweisung einen Wert zu berechnen.
Ausdruck und Wert haben also zwar miteinander zu tun, sind aber wesentlich verschiedene Dinge.

Arithmetische Ausdrücke

Etwas vereinfacht stellt sich ein Arithmetischer Ausdruck so dar:

```
<AAus> ::= ["+"|"-"] <Term> {<ArOp> <Term>}
<Term> ::= "("<AAus>")" | <Literal> | <Bezeichner> |
           "abs" "("<AAus>")" | <Funktionsaufruf>
<ArOp> ::= "+" | "-" | "*" | "/" | "mod" | "rem" | "** "
```

(die ganze Wahrheit steht im Ada-Reference-Manual, 4.4)

Dabei steht `x**y` für die Exponentation (x^y, $y \geq 0$) (für `y` ist nur `integer` bzw. `natural/positive` erlaubt). Ebenfalls nur für `integer` sind `mod` und `rem` definiert.

In dem Paket `Ada.Float.Elementary_Functions` stehen Funktionen für `float`-Zahlen wie Wurzel (`Sqrt`), Logarithmus (`Log`) und Trigonometrische Funktionen zur Verfügung (Details im ARM A.5.1).

Prioritäten der gängigen Operatoren in Ada 95

1. `abs, **`
2. `*, mod, /, rem`
3. `+, -` (als Vorzeichen)
4. `+, -` (als Addition/Subtraktion)

Gewisse Kombinationen, inbesondere solche, die für Normalsterbliche „mathematisch nicht eindeutig sind“, sind verboten und müssen mit Klammerungen eindeutig gemacht werden, z.B. `x/-y`, `x**y**z`, `x**abs(y)`. (Warum ist `x**abs(y)` nicht eindeutig? Selbst probieren und Fehlermeldung deuten ...)

Auch hier: Klammerungen erhöhen die Lesbarkeit!