Einführung in die Informatik I (autip)

Dr. Stefan Lewandowski

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

14. November 2007

Inhalte heute:

- Arithmetische und Boole'sche Ausdrücke
- Bäume oder wie wertet Ada Ausdrücke aus
- Subtypen
- Bereiche, Intervalle

Arithmetische Ausdrücke

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

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.

weiteres u.a. im Paket `Ada.Float.Elementary_Functions`
Prioritäten der gängigen Operatoren in Ada 95:
`abs, **` vor `*, mod, /, rem` vor `+, -`

Gewisse Kombinationen müssen mit Klammerungen eindeutig gemacht werden, vieles wird mit Klammerungen lesbarer!

Die Vergleichsoperatoren „=“, „/=“, „<“, „<=“, „>=“, „>“ haben den Ergebnis-Typ `Boolean` (entweder `true` oder `false`), wobei bei `float`-Zahl en die Vergleiche „=“, „/=“ aufgrund der unvermeidlichen Rundungsfehler mit Vorsicht zu genießen sind.

Boole'sche Ausdrücke:

```
<BAus> ::= <Rel> {<BoOp> <Rel>}
<Rel> ::= "true" | "false" | "not" <Rel> |
            "("<BAus>")" | <AAus><VglOp><AAus>
<BoOp> ::= "and" | "or" | "xor" | "and then" | "or else"
```

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

Als Besonderheit in Ada 95 gibt es keine Prioritätsregeln zwischen den Boole'schen Operatoren, insbesondere verlangt Ada 95 grundsätzlich eine Klammerung, wenn verschiedene Boole'sche Operatoren in einem Ausdruck vorkommen, so etwas wie `x and y or z` ist also verboten und muss geklammert werden zu `(x and y) or z` oder `x and (y or z)`.

Das `not` nimmt eine Sonderstellung ein und ist in der Priorität auf der Ebene von `**` und `abs` eingeordnet, d.h. `not abs(y) > 4` ist nicht erlaubt. (wie oben, selbst probieren)

Die Operatoren „`and then`“ sowie „`or else`“ erlauben eine verkürzte Auswertung von Boole'schen Ausdrücken. Z.B.

```
if x/=0 and then z/x > 10 then <Anweisung> end if;
```

oder

```
while x=0 or else z/x > 5 loop <Anweisung> end loop;
```

Bei `and`, `or` und `xor` werden in Ada 95 grundsätzlich beide Operanden ausgewertet.

Außer mit den logischen Operatoren können Boole'sche Variablen auch mit „`=`“, „`/=`“, „`<`“, „`<=`“, „`>=`“, „`>`“ verknüpft werden, wobei `false < true` gilt.

Wie wertet Ada 95 Ausdrücke aus?

Wir wollen hier nicht in die Tiefen des Compiler-Baus einsteigen, können aber die Gelegenheit nutzen eine grundlegende und sehr wichtige Struktur in der Informatik kennenzulernen: den Baum.

Betrachten wir hierzu einige Beispiele:

Definition von Bäumen:

- Im Allgemeinen kann in den Knoten der Bäume beliebiges stehen, in manchen Anwendungen interessiert auch nur die Struktur, so dass dann die Knoten unbeschriftet sind.
- Baumartige Strukturen spielen in der Informatik eine so wichtige Rolle, dass wir gleich die wichtigsten Begriffe kennen lernen sollten.
- **Definition:** Zur Definition eines Baums beginnt man interessanterweise besser mit dem Plural: Ein *Wald* ist eine (möglicherweise leere) Menge von Bäumen.
- Dann lässt sich leicht sagen, was ein Baum ist: ein *Baum* besteht aus einem Knoten (*Wurzel* genannt) und einem Wald der Unterbäume.

- Insgesamt haben wir also eine Menge von Knoten vor uns, zwischen denen gewisse Beziehungen bestehen. Zur Beschreibung dieser Beziehungen greift man auf eine Analogie zu familiären Beziehungen zurück: ein Knoten ist der *Vater* der Wurzelknoten seiner Unterbäume, die entsprechend *Söhne* heißen (es ist eine rein männlich Familie...).
- Knoten, deren Wald der Unterbäume leer sind, heißen *Blätter* (hier ist der Sprachgebrauch nicht einheitlich: manchmal wird die Wurzel als Blatt ausgeschlossen, bei uns ab er nicht).
- Da Elemente einer Menge keine feste Anordnung haben, haben wir auch keine Anordnung der Söhne eines Knotens; daher ist ein Rechenbaum kein Baum im Sinne dieser Definition (die Reihenfolge der Operanden z.B. in `x/y` ist wesentlich!). Das hindert uns aber nicht daran, das Vokabular sinngemäß auch für diese Struktur zu verwenden — spielt die Reihenfolge der Unterbäume eine Rolle, so spricht man von *geordneten Bäumen*.

Strenges Typkonzept mit Subtypen:

- Ada hat ein strenges Typkonzept, aber `integer` verträgt sich mit den Datentypen `natural` und `positive`?

  $\rightsquigarrow$ Die Datentypen `natural` und `positive` sind in Ada 95 Subtypen.
- Was ist ein Subtyp? Ein Subtyp ist eine Beschränkung eines Typs auf einen Teilbereich.
- Ein Beispiel: bei der Bearbeitung des Datums auf den letzten Übungsblättern, wäre es unter Umständen hilfreich gewesen, den Wertebereich für den Monat auf 1..12 einzuschränken. Dieses hätte man mit folgender Anweisung tun können:
  ```
  subtype Monatstyp is integer range 1..12;
  ```
  allgemein:
  ```
  subtype <Typname> is <datentyp> <constraint>;
  ```
- Auch hier gilt wieder das Prinzip der Les- und Wartbarkeit: Es ist gelegentlich praktisch, wenn Bereichsüberschreitungen automatisch als Fehler erkannt werden.

Strenges Typkonzept mit Subtypen (Folie 2):

- In Ada vordefiniert sind unter Anderem:
  ```
  subtype natural is integer range 0..integer'last;
  subtype positive is integer range 1..integer'last;
  ```
- Die Constraints werden in der Regel durch Einschränkung auf einen Bereich `range a..b` angegeben.
- Für den Subtypen stehen alle Operationen und Ausgabemöglichkeiten zur Verfügung, die auch der ursprüngliche Typ bereit stellt. Insbesondere können in Ausdrücken beliebige Subtypen eines Typs miteinander verarbeitet werden.
- Die Berechnungen finden innerhalb des Typs statt, nicht innerhalb des Subtyps.

Ein Beispiel:

- Mit obigem `subtype Monatstyp`, wäre z.B. folgende Deklaration denkbar:
  `monat : Monatstyp := 11+5-7;`
  Der Ausdruck würde als Integer-Ausdruck ausgewertet (dies trifft auch zu, wenn in dem Ausdruck nur Variablen eines Subtyps auftreten). Erst bei der Zuweisung wird überprüft, ob der Wert den Constraint erfüllt. Falls nicht, erzeugt das Programm einen Laufzeitfehler.

Bereiche:

- Bereiche (Schlüsselwort „`range`“) stellen Intervalle dar: hier gilt zunächst, dass `a..b` der Menge $\{x \mid a \leq x \leq b\}$ entspricht, insbesondere ist `a..b` leer, wenn `a` größer als `b` ist.

Bereiche/Intervalle und `for`-Schleifen:

Bereiche haben wir schon bei `for`-Schleifen kennengelernt. Dort wird der Schleifenvariablen bei jedem Durchlauf beginnend mit dem kleinsten Element jeweils das nächstgrößere Element des Intervalls zugewiesen. Ist der Bereich leer, so wird der Schleifenrumpf garnicht ausgeführt.

Will man die Elemente in umgekehrter Reihenfolge, also vom größten zum kleinsten, bearbeiten, so ist dies mit dem Schlüsselwort `reverse` möglich:

```
for i in reverse 1..5 loop
  put (i);
end loop;
```

gibt die Zahlen 5, 4, 3, 2 und 1 aus.

```
for i in 5..1 loop
  put (i);
end loop;
```

tut hingegen garnichts, da der Bereich `5..1` leer ist – es gibt keine Zahlen, die sowohl ≥ 5 als auch ≤ 1 sind.

Eine weitere Anwendung von Bereichen in Ada 95:

- Bereiche lassen sich auch in Boole'schen Bedingungen verwenden: in der Anweisung
  `if Zaehler in 3..9 then ...` würde der `then`-Zweig genau dann ausgeführt, wenn `Zaehler>=3 and Zaehler<=9` gilt.