Programmierkurs I

11.11.2002

Gliederung:

1. Deklarationsteil als BNF
2. Blöcke in Ada95
   (Lebenszeit, Sichtbarkeit von Variablen)

EBNF: Wiederholung

Die EBNF (Erweiterte Backus-Naur-Form) wurde in der Vorlesung (Abschnitt 1.1.6) zur Beschreibung der Ada-Syntax eingeführt. Wir verwenden im folgenden die auf Folie 228 eingeführte Schreibweise, d.h.

- Nichtterminale ohne spitze Klammern
- Terminalzeichen in Anführungszeichen ("...")
- Schlüsselwörter in Apostrophen ('...')
- beliebig wiederholbare Teile in geschweiften Klammern ({...})
- optionale Teile in eckigen Klammern ([...])

Beispiel:

if_statement ::= '`if`' condition '`then`' sequence_of_statements
{ '`elsif`' condition '`then`' sequence_of_statements }
['`else`' sequence_of_statements] '`end if`' ";"

EBNF: Wiederholung II

- Ein senkrechter Strich bedeutet Alternative:

  statement ::= {label} simple_statement
  | {label} compound_statement

- Manche Nichtterminale enthalten kursiv geschriebene Teile. Diese sind Hinweise; nur das restliche Nichtterminal tritt als linke Seite einer EBNF-Regel auf. Beispiel:

  condition ::= *boolean*_expression

  Der Zusatz *boolean* besagt, dass der Ausdruck in expression ein Wahrheitswert sein soll.

  Die komplette Syntax von Ada lässt sich in EBNF formulieren. Sie finden eine solche Darstellung z.B. auf

  `http://www.adahome.com/rm95/rm9x-P.html`

  (s.a. Folie 234).

Deklarationsteil

Der Teil eines Programms, in dem Variablen vereinbart (deklariert) werden, heißt *Deklarationsteil*; in den Programmen, die Sie bisher kennengelernt haben, tritt er zwischen

`procedure` *xxx* `is`

und `begin` auf.

In der Ada-Syntaxbeschreibung heißt er declarative_part.

Im folgenden wird die Syntax auf die für Sie zunächst wichtigsten Fälle reduziert. Wer mehr wissen will, kann die komplette Syntax nachlesen – siehe vorige Folie.

(1) declarative_part ::= { declarative_item }

(2) declarative_item ::= object_declaration | type_declaration
| subtype_declaration | …

(3) object_declaration ::= defining_identifier_list ":"
subtype_indication [":=" expression] ";"
| defining_identifier_list ":"
array_type_definition [":=" expression] ";"

(4) defining_identifier_list ::= *defining*_identifier
{ "," *defining*_identifier }

In (3) werden Variablen deklariert und mit dem Ergebnis des Ausdrucks in expression initialisiert; der Typ von expression muss mit subtype_indication kompatibel sein.

Deklarationsteil II

subtype_indication gibt den Typ der Variablen an. Einige vordefinierte Typen sind ‘`Integer`’, ‘`Float`’, ‘`Character`’, oder ‘`Boolean`’. Beispiele für Deklarationen nach (3):

```
x,y :  integer := 0;
  z :  character;
```

Weitere Typen können definiert werden, etwa als Untertypen oder Aufzählungstypen. Für erstere ist eine subtype_declaration zuständig:

(5) subtype_declaration ::= ‘`subtype`’ *defining*_identifier
‘`is`’ subtype_indication “;”

(6) subtype_indication ::= *subtype*_identifier [range]

(7) range ::= ‘`range`’ expression “..” expression

In (6) muss es sich bei *subtype*_identifier um einen bereits bekannten Typen handeln. In (7) müssen die expressions den in (6) angegebenen Typ haben.

Beispiel:

```
subtype ziffer is integer range 0..9;
```

Hier wird `ziffer` als Untertyp von `integer` vereinbart, der die Zahlen von 0 bis 9 enthält.

Deklarationsteil III

Aufzählungstypen können in einer type_declaration definiert werden:

(8) type_declaration ::= '`type`' *defining*_identifier
'`is`' type_definition ";"

(9) type_definition ::= enumeration_type_definition | ...

(10) enumeration_type_definition ::=
"(" *defining*_identifier { "," *defining*_identifier } ")"

Beispiel:

```
type tuer is (auf,zu);
```

Durch ihr Auftreten in dieser Deklaration werden `auf` und `zu` sogleich als Werte vereinbart, die einer Variablen vom Typ `tuer` zugewiesen werden können.

Felder

In Regel (3) war auch die Deklaration von Feldern zugelassen. Diese werden wie folgt deklariert:

(11) array_type_definition ::= ‘`array`’
“(” index_type { “,” index_type } “)”
‘`of`’ subtype_indication

(12) index_type ::= subtype_indication | range

Eine Nebenbedingung bei (12) ist, dass nur *diskrete* Typen verwendet werden dürfen, etwa Untertypen von Integer oder Aufzählungstypen, nicht aber z.B. Float.

Beispiele:

```
a:array (ziffer) of tuer;
b:array (0..5,0..3) of integer;
```

Felder können übrigens auch als Typen vereinbart werden; Ergänzung zur Regel (9):

(9) type_definition ::= enumeration_type_definition
| array_type_definition | …

Blöcke

Anweisungen in Ada können einfach oder zusammengesetzt sein.

(13) statement ::= {label} simple_statement
| {label} compound_statement

Unter einfachen Anweisungen verstehen wir z.B. Zuweisungen an Variablen, unter zusammengesetzten solche wie if, while etc. Eine besondere zusammengesetzte Anweisung ist ein Block, der eigene Deklarationen enthalten kann:

(14) compound_statement ::= … | block_statement | …

(15) block_statement ::= [*block*_identifier ":"]
['`declare`' declarative_part]
'`begin`' sequence_of_statements '`end`'
[*block*_identifier]

Dabei können unter declarative_part alle Arten von Vereinbarungen getroffen werden, wie sie laut den Regeln (1)–(12) zulässig sind. Diese Vereinbarungen (von Variablen, Typen etc.) gelten dann in dem durch '`begin`' und '`end`' eingeschlossenen Programmteil.

Blöcke II

Wird ein block_statement ausgeführt, so werden alle unter '`declare`' aufgezählten Deklarationen *elaboriert*, wobei u.a. die dort vereinbarten Variablen erschaffen (und ggf. initialisiert) werden. Vereinbarungen in inneren Blöcken *überdecken* die in äußeren Blöcken. Ist der Block vollständig ausgeführt, so werden seine Variablen wieder zerstört.

Beispiel:

```
procedure beispiel1 is
  i:integer;
begin
  for k in 1..5 loop
    i:=k;
    declare i:integer := 0;
    begin
      i:=i+1;
      put("inneres i=");put(i);new_line;
    end;
    put("auesseres i=");put(i);new_line;
  end loop;
end;
```

Blöcke III

Das Beispiel zeigt folgendes:

- Das Programm hat zwei verschiedene Variablen namens `i`, eine “äußere” (durch die Prozedurdeklaration) und eine “innere” (durch die Blockdeklaration).

- Die Anweisung `put(i)` innerhalb des Blocks bezieht sich auf das innere `i`, da seine Deklaration das äußere `i` überdeckt. Nur die zweite `put`-Anweisung bezieht sich auf das äußere `i`. Man sagt auch, das innere `i` sei innerhalb der Block-Anweisung *sichtbar*, das äußere `i` in der gesamten Prozedur mit Ausnahme der Block-Anweisung. Genauer: *direkt sichtbar*, aber dazu irgendwann mehr.

- Bei jedem Durchlauf der Schleife wird das innere `i` neu erschaffen und am Ende wieder zerstört. Die Ausgabe für das innere `i` ist also jedes Mal `1`. Demgegenüber “lebt” das äußere `i` für die gesamte Dauer des Programms; seine Ausgabe ist `1`,`2`,`3`,`4`,`5`.

Blöcke IV

Da Deklarationen zur Laufzeit elaboriert werden, können die darin enthaltenen Ausdrücke (expressions) auf Variablen Bezug nehmen, die in äußeren Blöcken vereinbart wurden. Es ist z.B. möglich, folgende Dinge zu schreiben:

```
procedure beispiel2 is
  j,k,n:integer;
begin
  ...
  declare
    i:integer:=j+k;
    a:array(0..n) of boolean;
  begin
    ...
  end;
  ...
end;
```

Dabei realisiert die Deklaration von `a` ein Feld mit einer variablen Anzahl von Einträgen.