stichting
mathematisch
centrum

MC

AFDELING NUMERIEKE WISKUNDE NN 32/83 DECEMBER

J. KOK

ADA VERGELEKEN MET PASCAL

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, Kruislaan 413, Amsterdam, The Netherlands.

The Mathematical Centre, founded 11 February 1946, is a non-profit institution for the promotion of pure and applied mathematics and computer science. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

69-01

1980 Mathematics Subject Classification: ~~68-01~~, 69D49.

Ada *) vergeleken met Pascal

Ada compared to Pascal

door

Jan Kok

SAMENVATTING

Een deel van de concepten van de programmeertaal Ada is snel te leren met de programmeertaal Pascal als voorkennis. De structuur van programma's en een aantal programma-elementen, zoals statements, declaraties, data-types en subprogram-declaraties, worden voor Ada vergeleken met die in Pascal.

TREFWOORDEN: Ada, Pascal, programmeertaal.

*) Ada is a registered trademark of the US Government (AJPO).

INHOUD

1. INLEIDING

Er bestaat een eenvoudige deelverzameling van de programmeertaal Ada, waarvan geldt dat de mogelijkheden veel op die van Pascal lijken, en er is dan ook een grote overeenkomst tussen de beide uitdrukkingsmiddelen. Hierin weerspiegelt zich de afkomst van Ada. Hieronder wordt ingegaan op deze overeenkomsten, waarbij gewezen wordt op de verschillen in betekenis van uiterlijk gelijke teksten in Ada en Pascal.

Vooraf: commentaar in Ada-teksten wordt als volgt gegeven: het begint met -- (twee mintekens) en duurt tot het eind van de regel; voor commentaar over meerdere regels moet dus op elke regel weer -- staan.
Verder: een 'identifier' is net als in Pascal een rij letters en cijfers waarvan het eerste symbool een letter is. Maar in Ada mag tussen twee opeenvolgende letters of cijfers van een identifier (of tussen een letter en een cijfer, enz.) een _ ('underscore') staan (dat mag in sommige Pascal-dialecten ook), en die _ is significant (dat is bij die Pascal-dialecten meestal niet zo). Bovendien zijn alle symbolen van een identifier significant, dus tot willekeurige lengte van de identifier, alleen wordt geen onderscheid gemaakt tussen hoofdletters en kleine letters. Lay-out binnen identifiers is niet toegestaan.
De notatie van getallen wordt bij Declaraties behandeld (zie hoofdstuk 3.a).

Net als in Pascal heeft in Ada een eenvoudig programma (d.w.z. een op zich zelf staande tekst die niet refereert naar externe modulen) een regelmatige structuur:

```
procedure MAIN is -- een programma is een 'subprogram'
                  -- is een procedure (hier zonder parameters)
   <declaraties>
begin
   <statements>
end MAIN;
```

Voor Pascal is dit

```
PROGRAM p ( file-parameters );
   <declaraties>
BEGIN
   <statements>
END.
```

Op het eerste gezicht is de structuur hetzelfde, en die indruk is juist, al is het niet waarschijnlijk dat het Ada-programma zonder een zgn. 'context clause' veel zin heeft. In nog niet behandelde Ada-terminologie doet een 'context clause' hier het volgende: hij staat het gebruik toe van een bibliotheek-onderdeel toe, nl. een 'package', dat procedures bevat om uitvoer te kunnen geven.

Misschien het meest opvallende verschil zal voor eenvoudige programma's zijn, dat in Ada een array-index door ronde, i.p.v. door vierkante haakjes omgeven wordt. Daarna blijkt, dat er veel termen in beide talen voorkomen. Vervolgens, dat ze vaak net iets anders en soms iets heel anders betekenen. Zie maar:
type definition, block (-statement), function designator,
compound statement,

procedure declaration en function declaration.

Syntax-regels leveren vaak wel teksten die op elkaar lijken, maar de productie-regels zijn soms erg verschillend:
Voor Pascal leren we dat tussen twee opeenvolgende statements een ; moet staan. Bij Ada onthouden we het gemakkelijkst dat na elke statement een ; staat, maar dat is syntactisch niet helemaal correct, omdat de ; nog tot de statement behoort.

De volgende onderwerpen zullen besproken worden:
de meeste statements,
declaraties,
type-definities,
expressies,
subprogram-declaraties en aanroepen,
in beperkte mate context clauses,
eenvoudige I/O.

Buiten het bestek van deze behandeling vallen (door afwezigheid van een Pascal equivalent):
packages, tasks, subunits, generics, exceptions, overloading,
representatie-specificatie, fixed-point types.

De gebruikte referenties zijn Pascal User Manual and Report, en Reference Manual for the Ada Programming Language. Er wordt meestal niet in detail naar verwezen.

2. STATEMENTS

De volgende statements komen in beide talen voor en zijn qua werking vergelijkbaar. Belangrijke verschillen worden aangeduid.

2.a. Lege statement

In Pascal mogelijk door niets te geven, in Ada moet er expliciet null; staan.

2.b. Assignment-statement en expressie

De assignment-statement is in beide talen van de vorm:

```
T-'variable' := T-'expression'
```

(in Ada met de noodzakelijke afsluitende ;).
Voor beide talen moet de expressie een waarde leveren van (een subtype van) het type van de variable in het linkerlid, waarbij de variable zelf ook wel van een subtype (van het expressie-type) mag zijn. Overigens is in Pascal een subrange (de enige manier om een subtype te maken) altijd een nieuw type, dat wel in assignments met zijn basis-type gemengd mag worden, terwijl bij Ada een subtype geen nieuw type is (maar altijd het subtype van een type), terwijl twee afzonderlijke types nooit gemengd mogen worden (ze zijn verschillend als ze uit aparte type-definities komen).
In Pascal is de assignment ook in gebruik voor het aan een functie-naam toekennen van het door de functie af te leveren resultaat, Ada heeft hiervoor de return-statement.
In Ada mag de variable zelfs een component (of alle componenten) van een door een functie-aanroep geleverde access-waarde (pointer-waarde) zijn, wat in Pascal niet direct mag (wel door eerst het functie-resultaat aan een pointer-variable te assigneren).
Een extra in Ada is dat een 'slice' van een 1-dimensionale array-variable, zeg A met indices 4 t/m 21, ook als variabele en expressie te gebruiken is, zodat bijv. A(5 .. 7) := expressie; legaal is, mits de expressie zelf een array-waarde van drie componenten van het zelfde componenten-type is, met integer indices. Bijv.:

```
A(5 .. 7) := A(14 .. 16); is toegestaan.
```

Een andere vorm om een waarde van een samengesteld type te krijgen is de 'aggregate' voor array- en record-type-waarden. In Pascal is iets dergelijks alleen bekend van het nonstandaard VALUE deel. Bijv.:

```
(4, 3, -8, 0, -7, 9)
(4 => 6, 5 .. 8 => 1, others => 0)
(DAG => 24, MAAND => NOVEMBER, JAAR => 1983)
```

Voor het maken van formules is veel overeenkomstig, maar Ada staat operatoren met operanden van verschillend type en van zelfgemaakt type niet automatisch toe. Dit gebruik mag als de operatoren daar speciaal voor gedefinieerd zijn, zelfs voor niet-numerieke types en samengestelde types (d.m.v. een operator-declaratie), waardoor uiteindelijk met Ada veel meer kan dan in Pascal denkbaar is.
Veelal zijn de problemen bij gemengd-type expressies op te lossen met expliciete type-conversies, bijv.

```
FLOAT1 ( FLOAT2-expressie )
```

die het resultaat geschikt maakt om aan een FLOAT1-variable of bij een

FLOAT1-operator te gebruiken. Ook voor de conversie van INTEGER naar FLOAT vice versa werkt dit, in het laatste geval met afronding.

Voor operatoren tenslotte gelden andere prioriteitsregels, en er zijn een paar extra, nl.:
and then, or else, xor, not in, &, rem, ** en abs
terwijl <> (in Pascal) in Ada /= heet; voor de (predefined) deling / geldt: het resultaat is reëel als de operanden reëel zijn, zijn beide operanden integer, dan is / de integer-deling (in Pascal DIV genaamd). rem is hetzelfde als mod (in Pascal: MOD) als alleen met positieve waarden gewerkt wordt (wat ik aanbeveel). & is de catenatie van strings, xor is 'exclusive or', and then en or else zijn de zgn. 'short circuit forms', waarbij de rechter operand gegarandeerd niet wordt uitgerekend als het resultaat op grond van de linker operand al vast staat.
De in en not in operator hebben wel een heel andere betekenis, bij Pascal is die of een bepaalde waarde in een verzameling-waarde voorkomt (zoals: 5 IN [0, 3 .. 6, 9, 12]), bij Ada is de rechter operand een (sub)type-naam (elk type mag) of een 'range' (een interval), maar NIET een gespreide verzameling waarden. In Ada is de betekenis: behoort de linker waarde (niet) tot het subtype rechts?
De prioriteitsklassen zijn van hoog naar laag:

1. ** abs not
2. * / mod rem
3. + - &
4. = /= < <= > >= in not in
5. and or xor and then or else

Bijzonderheden hierbij zijn nog:
De operatoren van de 5-e klasse mogen niet door elkaar voorkomen, dan moeten altijd haakjes gebruikt worden:
dus A or B or C or D is goed, maar
A or B and C or D is fout: er moet dan
((A or B) and C) or D
(A or B) and (C or D)
A or (B and (C or D)) of
A or (B and C) or D geschreven worden.
Voor ** geldt echter dat deze operator niet tweemaal achter elkaar gebruikt mag worden: niet A ** B ** C, wel: A ** (B ** C) of (minder zinvol): (A ** B) ** C.

2.c. Procedure-aanroep

De aanroep van een procedure kan in Ada net zo geschieden als in Pascal, maar in Ada kan meer ('named' parameter-associatie, 'default'-parameters), terwijl het parameter-doorgeef-mechanisme in Ada anders is (zie hoofdstuk 5.c en 5.e). Daar staat tegenover, dat een subprogram in Ada niet met procedures of functies geparametriseerd kan worden (tenzij statisch m.b.v. het 'generics' begrip), wat in officieel Pascal wel mogelijk is.

2.d. Goto-statement

Uiterlijk: goto 'label-naam'; (in Pascal: GOTO label-nummer).
Net als in Pascal mag in Ada niet binnenwaarts in een gestructureerde statement worden gesprongen. Het gebruik wordt in Ada nog meer

bemoeilijkt, doordat niet meer naar een label in een omvattend subprogram gesprongen mag worden. Labels worden niet gedeclareerd, ze staan a.v. voor een statement: << EEN_LABEL >> A := B ; (in Pascal is per statement maar 1 label toegestaan).

2.e. Block-statement en zichtbaarheid

De compound statement van Pascal (BEGIN s; s; s END) is in Ada-programma's veel minder nodig, omdat de meeste gestructureerde statements (die andere statements kunnen bevatten) meteen een rij statements mogen bevatten:

Pascal: WHILE conditie DO statement
Ada: while <conditie> loop <rij statements> end loop;

Toch heeft Ada een equivalent, dat het meteen mogelijk maakt lokale declaraties te doen, en een lokale 'exception-handler' te plaatsen. Bijv.:

```
declare
   <declaraties>
begin
   <rij statements>
-- misschien nog: exception <exception-handlers>
end;
```

of:

```
BLOCKNAAM:
   declare
      <declaraties>
   begin
      <rij statements>
   end BLOCKNAAM;
```

De blocknaam kan gebruikt worden voor het bereiken van een object of iets anders dat door een hergebruik van de naam in een declaratie of door dubbelzinnig-zijn van de naam niet bereikbaar (niet direct zichtbaar) is. Bijv.:

```
BUITEN :
  declare
     X : FLOAT; -- 1-e declaratie van X
  begin
     declare
        X : INTEGER := 3; -- 2-e declaratie van X
     begin
        X := X + 1; -- dit is altijd de integer X, de andere is niet
                    -- direct zichtbaar, maar wel omslachtiger:
        Y := 11.56 * BUITEN.X; -- verwijst naar de X van BUITEN.
     end;
  end BUITEN;
```

2.f. If-statement

De Ada if-statement heeft een afsluitend symbool: end if, waardoor zowel tussen then en else (of end if) als tussen else en end if meteen een rij statements mag staan, zonder dat daarvoor een extra begin en

end gebruikt moeten worden. Bovendien kan, indien de (enkele) statement tussen else en end if weer een if-statement zou zijn, dit samengetrokken worden met elsif, wat geen nieuwe end if vergt. De structuur is bijv.:

```
if <conditie> then
   <rij statements>
elsif <nieuwe conditie als de eerste FALSE geeft> then
   <rij statements>
elsif <nieuwe conditie, enz.> then
   <rij statements>
else
   <rij statements>
end if;
```

else met de daarop volgende rij statements mag ook afwezig zijn.
Dit dekt wel alle gevallen bij Pascal, terwijl enkele moeilijkheden (zoals bij THEN IF ..., met later 1 ELSE) niet meer voorkomen.

2.g. Loop-statement en exit-statement

Tegenover de drie vormen van repetitive statement van Pascal (for, repeat, en while statement) heeft Ada de loop-statement met drie mogelijkheden, nl.

```
de kale loop:          loop <rij statements> end loop;
de loop met for:       for I in L .. U loop < ... > end loop;
of:           for I in reverse L .. U loop < ... > end loop;
de loop met while:     while <conditie> loop < ... > end loop;
```

Verder kan een loop-statement nog een naam hebben, net als bij een block-statement, bijv. voor de kale loop:

```
LOOP_NAAM : loop <rij statements> end loop LOOP_NAAM;
```

In de rij statements mogen exit-statements voorkomen, die bij uitvoering meteen de loop beëindigen, wat vooral bij de kale loop wel nuttig is maar ook bij de andere vormen mag. Voorbeelden zijn:

```
exit;
exit LOOP_NAAM; -- bij geneste loops wel praktisch,
exit when <conditie>; -- hetzelfde als:
   if <conditie> then exit; end if;
exit LOOP_NAAM when <conditie>;
```

In de for-loop mag het interval L .. U ook een subtype-aanduiding voor een discrete (sub)type zijn (bijv. POSITIVE, of:
POSITIVE range 3 .. 30) of een range-attribute (bijv. A'RANGE als A een array-variable is).
Bovendien geldt (ook weer voor de for-loop) dat de 'loop parameter' (d.i. de lopende variabele van de for-loop) niet gedeclareerd mag en kan worden: het is automatisch een lokale constante van het door de range (bijv. L .. U) aangeduide subtype.

Hiermee vinden we voor de Pascal-vormen de volgende equivalenten:

```
voor: FOR i:= 1 TO u DO s
      for I in L .. U loop S; end loop;
```

```
voor: FOR i:= u DOWNTO 1 DO s
      for I in reverse L .. U loop S; end loop;
voor: WHILE conditie DO s
      while <conditie> loop S; end loop;
voor: REPEAT s; s; s UNTIL conditie
      loop S; S; S;
         exit when <conditie>;
      end loop;
```

2.h. Case-statement

Bij Pascal:

```
CASE expressie OF
   case-label, case-label, enz. : statement;
   case-label, case-label, enz. : statement;
   enz.
END
```

Hierin moeten de case-labels constanten van het (discrete) type van de case-expressie zijn, dus identifiers of notaties van constanten, al of niet met een teken. Er hoeven alleen case-labels te zijn voor waarden die de expressie actueel zal aannemen.
Voor Ada is de vorm van een case-statement:

```
case <expressie> is
   when <keuze(n)> => <rij statements>
   when <keuze(n)> => <rij statements>
   enz., met eventueel als laatste alternatief:
   when others => <rij statements>
end case;
```

Meerdere keuzen in 1 alternatief worden a.v. gegeven:

```
keuze | keuze | keuze
```

en elke keuze (die zowel een 'simple expression' als een 'discrete range' mag zijn) moet statisch zijn, d.w.z. terug te voeren tot constante-notaties, eenvoudige expressies met constanten, namen van waarden van een 'enumeration type', en intervallen van al deze dingen.
Bijv.:

```
-3 | 5 .. 8 | (3+4)*2 -- voor een integer case-expressie, of:
'A' .. 'F' | '0' .. '9' | ASCII.CR
                          -- voor een CHARACTER-expressie.
```

Bij Ada moeten de keuzen samen alle waarden van de (sub)type van de expressie dekken, wat gemakkelijk met het others-alternatief te bereiken is.

3. DECLARATIES

Net als in Pascal mag in Ada-programma's een zelfgekozen naam pas gebruikt worden nadat deze in een declaratie is verschenen (en de naam is weer onbruikbaar als een voor elke declaratie geldend zgn. declaratie-gebied wordt verlaten). Verder legt een declaratie het doel van het gedeclareerde ding vast, d.w.z. waar het wel en niet voor gebruikt mag worden.

In Pascal worden declaraties geplaatst aan het begin van een block (dat is: het interne deel van een (deel-)programma) in aparte rubrieken die met de symbolen LABEL, CONST, TYPE, en VAR beginnen, voor de declaratie of definitie van labels, constanten, types en variabelen. Daarna kunnen nog volgen (een VALUE deel en) procedure- en functie-declaraties.
Voor Ada mogen in een 'declarative part' de declaraties van verschillende dingen in elke gewenste volgorde staan, zolang iets maar niet gebruikt wordt voordat het gedeclareerd is. Er zijn geen speciale rubrieken en rubriektitels.

In dit hoofdstuk en hoofdstuk 4 komen aan de orde de declaratie van:
numbers, types, subtypes, constante en variabele objecten.
Het declareren van subprograms staat in hoofdstuk 5.

Het gebied van geldigheid van declaraties in een subprogram body of een block-statement is het stuk tekst tussen het eind van de declaratie en het eind van de betreffende subprogram body of het block-statement. Het geldigheidsgebied van declaraties in packages en tasks wordt hier niet behandeld, en voor de geldigheid van formele parameters en velden of discriminanten van record types zijn de regels wat ingewikkelder, net als bij Pascal. Bij afwezigheid van een subprogram-declaratie begint het geldigheidsgebied van de subprogram-naam bij het eind van de specificatie die het begin van de subprogram body vormt.

3.a. Number declaraties

Aan getal-notaties kan een naam (of meerdere) worden toegekend, zodat verder met die naam (of namen) gewerkt kan worden.
Daar dit los staat van de declaraties van integer en real types, wordt het type van getallen en dus ook van zgn. 'named numbers' 'universal integer' of 'universal real' genoemd. Bij gemengd gebruik van 'numbers' met de standaard types INTEGER en FLOAT of extra standaard types met een andere nauwkeurigheid of met zelf gemaakte getallentypes moet de voorgestelde waarde in de volle precisie beschikbaar zijn. De declaratie heeft de gedaante:

```
NAAM1, NAAM2, enz. : constant := <statische getal-expressie>;
```

Bijv.:

```
ONE, EEN, UN, EINS : constant := 1;
PI : constant := +3.14159_26536;
TWEE_PI : constant := 2.0 * PI;
```

De expressie mag alleen getal-notaties en 'named numbers' bevatten, en de bekende operatoren voor aritmetiek, met enkele extra gevallen voor gemengde integer-real operaties.
Getal-notaties kunnen zijn:
1. basis 10 getallen ('decimal literals'):
1.1 integer literals:

vorm: ddd of dddEddd of dddE+ddd
waarin: ddd: een rijtje cijfers (minstens 1 cijfer) en tussen twee opeenvolgende cijfers mag een ('1') _ ('underscore') staan.
Bijv.: 1_983 50E8 24E+64

1.2 real literals:
vorm: ddd.ddd of ddd.dddEddd of ddd.dddE+ddd of ddd.dddE-ddd
Bijv.: 0.002_5 3.0E8 0.314_159_26E+1 0.5E-9

D.w.z.: zodra er een decimale punt voorkomt is het een 'real literal', en een 'integer literal' mag alleen een niet-negatieve exponent hebben.

2. getallen met andere basis ('based literals'):

2.1 based integer literals:
ddd#uuu# of ddd#uuu#Eddd of ddd#uuu#E+ddd
Het rijtje cijfers ddd voor de # moet een getal tussen 2 en 16 zijn: de basis. uuu is een rijtje uitgebreide cijfers (weer mogelijk met _'s ertussen), d.w.z. zoveel cijfers en de letters A, B, C, D, E en F als voor de basis nodig zijn (hexadecimaal is het maximum). De exponent betekent nu: * (basis ** ddd).
Bijv.: 2#101_001# 16#7E#E+1

2.2 based real literals:
als based integer literals, maar met uuu.uuu en ook negatieve exponent.
Bijv.: 3#102.002# 16#7E.E7#E-1

In getalnotaties mag verder geen lay-out voorkomen.

Het verschil met getal-notatie in Pascal is, dat based numbers in Pascal (officieel) ontbreken, en dat in Pascal een getal met een exponent altijd real is. In Ada zijn notaties met een integer mantisse, dus zonder decimale punt, maar met een negatieve exponent, niet toegestaan.
De number-declaratie komt overeen met de constant-definitie van Pascal voor getallen, maar in Ada mogelijk met meer namen in 1 declaratie, en met een expressie voor de waarde.

3.b. Type declaraties

Type-declaraties hebben de vorm:
type T is <type definitie>;
(of: type T(<discriminant-specificaties>) is <type definitie>;).
Elke type-declaratie introduceert een nieuw type en Ada is (zoals al bij expressie en assignment gesteld is) erg streng over het door elkaar gebruiken van verschillende types, zelfs als de waarden van het type verwant zijn, bijv. bij twee integer types.
Hoe in Ada nieuwe types gemaakt kunnen worden wordt in hoofdstuk 4 behandeld. Aan de orde komen: kopieën van bekende types, nieuwe integer en floating-point real types, discrete types, array en record types, en access types.

3.c. Subtype declaraties

Subtype-declaraties hebben de vorm:
subtype S is <subtype-aanduiding>;
waarbij de subtype-aanduiding bestaat uit de naam van een (sub)type, desgewenst door een zgn. 'constraint' gevolgd.
Een subtype-declaratie geeft een nieuwe naam aan een deelverzameling

van de verzameling van alle waarden van een (sub)type, maar er wordt geen nieuw type geintroduceerd. Waarden en variabelen van het subtype kunnen in expressies en assignments voorkomen waar de zelfde dingen van het oorspronkelijke type toegestaan zijn.

De betekenis van de 'constraint' is:

1) (range constraint: range <een range>)
   Voor een aantal types, nl. van getallen en van discrete types, kan de verzameling van waarden ingeperkt worden tot een deelinterval van waarden door een constraint als: range 4 .. 8. Bijv.:
   subtype POSITIVE is INTEGER range 1 .. INTEGER'LAST;
2) (accuracy constraint)
   Voor real types kan de verzameling van waarden verkleind worden door een vermindering van de dichtheid van reële getallen op de reële as, nl. door:
   - een floating-point constraint (digits D) voor floating-point types, D moet een statische integer-expressie zijn;
   - een fixed-point constraint (delta D) voor fixed-point types, D moet een statische real-expressie zijn.
   (Statisch betekent: de expressie moet uit operatoren en operanden bestaan die in compile-tijd kunnen worden uitgerekend.)
   Bijv.: subtype DIGITS_5 is FLOAT digits 5;
3) (index constraint, discriminant constraint)
   Bepaalde gedeclareerde types worden 'unconstrained' genoemd: dat houdt in dat er nog een vrijheid is, nl. een array-declaratie bepaalt wel altijd wat het type van de componenten moet zijn, en dat de indices van een bepaald discreet type moeten zijn, maar niet per se wat de precieze grenzen zijn (al kan dat wel):
   type ART is array (INTEGER range <>) of CT;
   In deze vorm wordt een unconstrained array-type ART gedeclareerd (zie hoofdstuk 4.e). Met een zgn. index-constraint worden de index-grenzen wel vastgezet, bijv.:
   subtype SART is ART(1 .. 20);
   maar met dezelfde ART kunnen ook andere (vector-)subtypes gemaakt worden, die alle nog tot het zelfde type behoren. Dit rechtvaardigt ook de assignment (zie hoofdstuk 2.b): A(5 .. 7) := A(14 .. 16);
   De andere vorm van 'constraining' geldt voor record-types met een of meer discriminanten; dan kunnen met 'discriminant constraints' ge-'constrain'de subtypes gemaakt worden.
   Index- en discriminant-constraints mogen dus alleen op unconstrained array-, resp. record-types worden toegepast.

3.d. Object declaraties

In Ada zijn er drie vormen:

<identifier-list> : <subtype-aanduiding>;
   Bijv: X, Y, AFSTAND : FLOAT;
<identifier-list> : <subtype-aanduiding> := <expression>;
   Bijv: X, Y, AFSTAND : FLOAT := 0.0;
<identifier-list> : constant <subtype-aanduiding> := <expression>;
   Bijv: GRADEN, DEGREES : constant FLOAT := PI / 180.0;

In plaats van 'subtype-aanduiding' mag er in Ada ook een array-definitie staan, bijv.:

   V, W : array (INTEGER range 2 .. 12) of FLOAT;

De eerste vorm is analoog aan de Pascal variable-declaration, maar in Ada mag in de subtype-aanduiding na de (sub)type-naam nog een con-

straint staan, waarvan de betekenis is, dat er eigenlijk een (anoniem) subtype geintroduceerd wordt voor de variable, of verschillende subtypes voor elke variable uit de lijst (de 'identifier-list').
In Pascal is een type-naam niet verplicht, in de VAR-declaratie mag elk type in formule uitgeschreven worden.
De tweede vorm geeft meteen een initiële waarde aan de variabelen, telkens als het declaratie-deel 'geëlaboreerd' wordt (dat is bijv. elke keer dat een subprogram aangeroepen wordt of dat een block-statement uitgevoerd wordt). In Pascal bestaat dit niet. Een meervoudige declaratie moet wegens mogelijke zij-effecten in de subtype-aanduiding of in de initiële expressie als een reeks individuele declaraties opgevat worden. Bijv.:

```
A, B, C : INTEGER range I1 .. I2 := I3;
```

moet worden opgevat als:

```
A : INTEGER range I1 .. I2 := I3;
B : INTEGER range I1 .. I2 := I3;
C : INTEGER range I1 .. I2 := I3;
```

De derde vorm (declaratie van constante objecten) komt met de constante-definitie van Pascal overeen, alleen wordt in Ada het type van de constante opgegeven en het kan dan ook voor elk type, terwijl bij Pascal het type moet blijken uit de soort van constante waarde (dus alleen een getal, boolean, character, string, scalar constante, of NIL, en zeker geen expressie). In Ada kan de expressie bij elke elaboratie een andere waarde leveren (dit is van toepassing bij meervoudige constante-declaraties, maar ook bijv. bij herhaalde aanroep van het deelprogramma dat de declaraties bevat).

4. TYPE DEFINITIES

Type-definities volgen op is in een type-declaratie (zie hoofdstuk 3.b) en elke type-definitie vormt een nieuwe type.
De mogelijkheden zijn:

4.a. Derived types

De gedaante van een declaratie van een derived type DT is:

```
type DT is new T; -- eventueel na de (sub)typenaam T een constraint.
```

Het type DT is een kopie van T, en DT erft alle deelprogramma's (predefined en user-defined), die al voor T aanwezig zijn. DT en T zijn echter verschillende types, die onderling alleen uitwisselbaar zijn door een expliciete type-conversie (zie hoofdstuk 2.b).
In Pascal lijkt dit op:

```
TYPE dt = t;
```

maar dit maakt alleen een synoniem van t. In Ada wordt het symbool type aan het begin van elke type-declaratie herhaald.

4.b. Enumeration types

De vorm is:

```
type ET is (<enumeral>, <enumeral>, enz. );
```

waarin een enumeration-literal of enumeral een identifier is of een character-notatie. Bijv.:

```
type KLEUREN is (ROOD, BLAUW, GEEL, GROEN, ZWART);
type CIJFERS is ('8','3','1','9','0', '2','4','5','6','7');
                -- in alfabetische volgorde.
```

Dit lijkt op de scalar types van Pascal, maar daar mogen geen character-notaties in gebruikt worden. Bovendien kan de waarde in Ada naar string geconverteerd worden (de functie-aanroep KLEUREN'IMAGE(BLAUW) levert "BLAUW").
BOOLEAN en CHARACTER zijn nu speciale gevallen (predefined) van een enumeration type.

4.c. Integer types

Integer types kunnen in een programma gemaakt worden, in een integer type definitie, die de vorm heeft van een 'range constraint' met statische grenzen, bijv.:

```
type SMALL_INT is range -5 .. +5;
```

Een implementatie zal beperkingen opleggen betreffende de mogelijke waarden van de grenzen, maar het is denkbaar dat het interval van waarden groter is dan dat van INTEGER; naar verwachting zal er dan een van INTEGER verschillend 'predefined' type LONG_INTEGER zijn.
Een zelfgemaakt type als SMALL_INT erft geen operaties van INTEGER (maar de 'predefined' integer operaties gelden er wel voor).
Zoals al is opgemerkt komt de subrange-definitie (voor integers) van Pascal (bijv.: TYPE subr = -5 .. +5;) meer overeen met een subtype in Ada, hoewel hij uiterlijk het meest met de integer type-definitie overeenkomt.

4.d. Real types

Analoog aan de zelfgemaakte integer types d.m.v. een integer type definitie zijn ook real types te maken d.m.v. een real-type definitie, die een 'floating-point constraint' of een 'fixed-point constraint' kan zijn. De eerste bestaat uit:

digits <statische integer-expressie>

eventueel gevolgd door een range constraint, de tweede uit

delta <statische real-expressie> range <statische range>.

Het verschil met de standaard real types (m.n. FLOAT) is analoog als bij integer types. Deze manier van real types maken heeft geen equivalent in Pascal.

4.e. Array types

Zoals al in hoofdstuk 3.c.3) is aangekondigd kunnen in Ada zgn. 'unconstrained' en 'constrained' array types gedefinieerd worden.
De gedaante van een unconstrained array definitie is:

array (type-mark range <>, type-mark range <>, enz.) of
    component-subtype-aanduiding.

Een index-subtype-definitie (of meerdere in geval van een meer-dimensionaal array) van de vorm 'type-mark range <>' geeft met een (sub)-type-naam aan wat het type van de index-expressie(s) moet zijn, als componenten van een array-object geselecteerd moeten worden, of wat het type van de grenzen in een index-constraint is, als een object gedeclareerd of een constrained subtype gemaakt moet worden. Bijv.:

```
type VEC is array (INTEGER range <>) of FLOAT;
X, Y : VEC (3 .. 6) := (-1.0, 0.0, 100.5, - PI);
Z : VEC (-3 .. 445);
subtype VEC_6 is VEC(1 .. 6);
```

De objecten X, Y en Z zijn van het zelfde type (al zijn ze van drie verschillende subtypes).

Men kan ook meteen een constrained array type definieren:

array (index-constraint, index-constraint, enz.) of
    component-subtype-aanduiding.

Een index-constraint (of meerdere in geval van een meer-dimensionaal array) leidt tot een (sub)type-naam (mogelijk nog gevolgd door range <range>) of meteen tot <range>. In het laatste geval kan het type van de index (of indices) onzeker zijn, nl. als de 'simple expressions' van de range uit getal-notaties of 'numbers' opgebouwd zijn.
Bij voorbeeld:

```
type VEC_6 is array (1 .. 6) of FLOAT;
P, Q : VEC_6;
R : array (SMALL_INT) of BOOLEAN;
```

Object P hierboven en object X in het vorige voorbeeld zijn wel van verschillend type, maar hun waarden zouden wel naar elkaar geconvertéerd kunnen worden als de lengten gelijk waren. Bijv.:

```
X := VEC( P(1 .. 4) );
```

Verder moeten voor het object R de index-expressies van het type

SMALL_INT zijn, bij P en Q is het index-type anoniem (integer).
In plaats van integer types kunnen voor de index definities ook (zelfgemaakte) enumeration types gebruikt worden.

We merken op dat array-objecten altijd van een constrained (sub)type moeten zijn.

Pascal kent het onderscheid tussen 'unconstrained' en 'constrained' niet, alle array types zijn meteen constrained. Bovendien zijn de grenzen in een array type definitie in Pascal constanten, misschien het grootste bezwaar van Pascal. Evenals in Ada mogen in Pascal de index-types 'scalar types' zijn. In Pascal mag na ARRAY [....] OF elke type-definitie staan, in Ada is alleen een (sub)type-naam plus een constraint mogelijk.
Een belangrijk gevolg van het 'unconstrained' zijn van array types is, dat subprograms gemaakt kunnen worden die met array-objecten van verschillende lengte als actuele parameter aangeroepen kunnen worden. Overigens kan in Ada met 'constrained' array-types ook op de Pascal-manier geprogrammeerd worden.

Voor het selecteren van een component van een array-object wordt in Ada net als in Pascal achter de object-naam tussen haakjes een index-expressie gegeven, of zoveel index-expressies als de dimensie van het object bedraagt. Elke index-expressie moet een waarde leveren in het betreffende index-interval. Zoals in de inleiding al is opgemerkt: opvallend is dat die haakjes ronde haakjes moeten zijn, terwijl vergelijkbare talen daar vierkante haakjes voor hebben.

Het reserved word PACKED van Pascal heeft geen direct equivalent in Ada. Wel is met het pragma PACK of met 'representation clauses' een bepaalde wijze van zuinig inpakken af te dwingen (als het al niet automatisch zo zuinig mogelijk gaat in Ada), wat hier verder niet behandeld wordt. In plaats van de Pascal procedures 'pack' en 'unpack' kan van de (al besproken) type-conversie gebruik gemaakt worden.

Zonder enige extra PACK-opdracht is het 'unconstrained' 1-dimensionaal array-type STRING beschikbaar, met index-type het subtype POSITIVE van het type INTEGER, en met component-type CHARACTER. Elke string-notatie, dat is een rij symbolen omgeven door 'quotes' ("), is een waarde van dit type, van het subtype dat bij zijn lengte hoort.

4.f. Record types

Net als bij array-types kunnen er zowel 'unconstrained' als 'constrained' record-types gedefinieerd worden, maar de eerste vorm (waarin de record-type-declaratie (een) discriminant specificatie(s) bevat) zullen we hier alleen kort schetsen. De record-type-definitie heeft de vorm:

```
record
   <component list>
end record
```

Afgezien van de herhaling van record na het end lijkt dit veel op de Pascal-definitie. De 'component list' (die ook alleen null; mag luiden) definieert de velden van het record. Het vaste deel van een record type bestaat uit component-declaraties, die in zoverre op gewone object-declaraties lijken dat er weer geen type-definitie in verstopt mag

worden, wel mag er een initialisatie in zitten. Bijv.:

```
record
   RE, IM : FLOAT := 0.0;
end record
```

of:

```
record
   NAAM, STRAATNAAM : STRING(1 .. 20);
   HUISNUMMER : POSITIVE;
end record
```

Na of in plaats van dit vaste deel mag nog een zgn. 'variant part' volgen (net als in Pascal) waarvan de vorm is:

```
case <discriminant-identifier> is
   when <keuze> | etc. => <component list>
   when <keuze> | etc. => <component list>
   etc.
end case; -- (dus een end meer dan in Pascal)
```

In dit geval is er sprake van een unconstrained record-type, en de discriminant van de variant moet aan het begin van de hele type-declaratie in een discriminant-specificatie (zie 3.b) gegeven zijn. Overigens kunnen de discriminanten van een record-type ook voor andere variaties dan de hier gedemonstreerde record-variant gebruikt worden. Bijv.:

```
type KLEUREN is (ROOD, ZWART, GROEN, BLAUW, WIT);
type REC_T (D : KLEUREN) is
   record
      NUM : INTEGER := 0;
      case D is
         when ROOD | WIT => NAAM : STRING(1 .. 20);
         when ZWART .. BLAUW =>
            TEMP : REAL; SYM : CHARACTER;
      end case;
   end record;
```

De overeenkomst met Pascal zal ook hier duidelijk zijn, een verschil is in Ada de vermelding als record-discriminant van de variant-selector.

Er is geen Ada-equivalent van de WITH statement van Pascal: als het werken met een bepaalde component van een record-object erg vaak voorkomt zou een lokale 'renaming-declaratie' wel nuttig kunnen zijn (deze wordt hier niet behandeld).

4.g. Access types

Access-types in Ada komen overeen met pointer-types in Pascal. De declaratie in Ada heeft de vorm:

```
type ACCESS_TYPE is access DESIGNATED_TYPE;
```

Om (net als in Pascal) recursieve types te kunnen maken, mag in Ada eerst een 'incomplete type declaration' van het aangewezen type gegeven worden, terwijl na de declaratie van het access-type de volle declaratie volgt die het aangewezen type gebruikt. Bijv.:

```
type CELL; -- incomplete
type LINK is access CELL;
type CELL is -- nu complete
   record
      VALUE : INTEGER;
      SUCC, PRED : LINK; -- gebruik van LINK
   end record;
```

Nieuwe aangewezen objecten (op de HEAP) worden verkregen door een zgn. 'allocator' (new ....) in een assignment aan een access-variabele:

```
POINTER : LINK; -- nu assignments als: POINTER := new CELL;
   -- of met initialisatie van POINTER:
POINTER : LINK := new CELL;
   -- of met tevens initialisatie van POINTER.all, dat is de
   -- aangewezen variable:
POINTER : LINK := new CELL'(0, null, null);
```

In de eerste initialisatie staat na new een 'subtype aanduiding', en wel het bij het access-type behorende type, in het tweede geval staat er een 'qualified expression', waarvan het meegegeven aggregate gelijk de aangewezen variable initialiseert.
Als POINTER niet null is, is POINTER.all de record-variable die POINTER aanwijst (in Pascal: p↑). Maar all kan weggelaten worden als meteen na POINTER een record-component geselecteerd wordt:
POINTER.VALUE voor POINTER.all.VALUE (in Pascal p↑.value).
Als een access-object declaratie geen initialisatie bevat wordt de variable op null geinitialiseerd. Verder kan met access types op dezelfde wijze gewerkt worden als in Pascal.
Het wordt algemeen betwijfeld of elke Ada-implementatie een efficiente 'garbage collector' zal krijgen om automatisch niet meer gebruikte geheugenruimte in de HEAP te recyclen. Gesuggereerd wordt dat een programmeur zelf zo zuinig mogelijk met via allocators verkregen HEAP-ruimte omspringt. Bepaalde pragma's zouden dat ondersteunen.

5. PROCEDURES EN FUNCTIES

Voor het gestructureerd opbouwen van programma's parallel aan het ontwikkelen van een oplossingsmethode, waarbij een groot probleem eerst wordt onderverdeeld in deelproblemen, kent Ada net als Pascal het definieren van deelprogramma's en het aanroepen ervan. In Pascal kunnen deelprogramma's (procedures en functies) voor de uitvoering van een deeltaak als ware het een enkele opdracht in elk declaratie-deel gedeclareerd worden na de eventuele overige declaraties (van constanten, types, variabelen). Activatie geschiedt door een procedure-aanroep als statement of een functie-aanroep als factor van een expressie op een plaats waar de declaratie zichtbaar is (of in bijzondere gevallen doordat het deelprogramma als parameter is doorgegeven aan andere deelprogramma's).
Wat in Ada gedaan kan worden lijkt hier veel op, zij het dat er ook andere dingen mogelijk zijn. We gaan hier niet in op de mogelijkheid dat deelprogramma's als zelfstandige bibliotheek-modulen aangemaakt kunnen worden. Op de Pascal-manier kunnen deelprogramma's in elk declaratie-deel (dus genest) gedefinieerd worden. We vermijden hiervoor het woord 'declaratie': het Pascal-equivalent van een procedure- of functie-declaratie is in Ada een 'subprogram body', maar er is in Ada ook een zgn. 'subprogram declaration', die niet na maar tussen de andere declaraties van een declaratie-deel (maar ook tussen de bodies) mag voorkomen, en die alleen de externe specificaties van de later volgende body bevat, dat zijn de naam, de formele parameter-lijst en eventueel het type van het resultaat van een functie. Deze splitsing van body en declaratie is te vergelijken met de forward-declaratie van Pascal, maar in Ada moet bij de body de specificatie van de parameters weer volledig herhaald worden (zie 5.a).

In de volgende secties komen aan de orde:
plaats en wijze van definieren van deelprogramma's,
soorten deelprogramma's,
externe specificatie: parameters en functie-resultaat,
interne specificatie: lokaal declaratie-deel en statement-deel,
aanroep en parameter-associatie.
Bij de behandeling van de soorten deelprogramma's wordt wel iets over 'overloading' (meer declaraties voor de zelfde naam) gezegd, maar veel over mogelijke dubbelzinnigheden en oplossing hiervan valt buiten het bestek van deze behandeling.

5.a. Plaats en wijze van definieren van deelprogramma's

In een declarative part van een subprogram body of van een block-statement kunnen lokale deelprogramma's gedefinieerd worden, door een subprogram body ervoor in het 'tweede' deel van dat declarative part te geven (het 'eerste' deel bevat alleen declaraties, bodies vormen het eventuele tweede deel, samen met extra subprogram declaraties, maar geen declaraties van bijv. objecten of types).
Naast deze eenvoudige wijze van definieren is het ook mogelijk om eerst een subprogram-declaratie te geven, later (in het zelfde declarative part) gevolgd door een bijbehorende body: de eerdere declaratie bestaat louter uit de externe specificatie, die in de body weer helemaal herhaald wordt. Zulke declaraties kunnen tussen de overige declaraties staan, maar ook tussen (andere) bodies. Deze splitsing kan gebruikt worden om een subprogram-aanroep (in een andere body) te kunnen geven

voordat de body van het aangeroepen subprogram zelf gegeven is (analoog aan de forward declaratie van Pascal). Een andere reden voor deze mogelijkheid is dat in package declaraties (het zichtbare deel van packages) alleen specificaties van subprograms mogen staan, de bodies staan dan in de bijbehorende onzichtbare package bodies.
Subprogram bodies kunnen wel zonder een expliciete subprogram declaratie voorkomen, het omgekeerde mag niet.
Bijv.:

```
declare                                  -- een block-statement
   type A is ....;
   procedure INIT(X : out A); -- alleen declaratie.
   B, C : A;

   function GET_A(J : INTEGER) return A is -- dit is
      L : A;                               -- een hele function body
   begin
      INIT(L); -- aanroep van bekende procedure INIT
      ....
      return L;
   end GET_A;

   procedure INIT(X : out A) is -- specificatie herhaald.
   begin                        -- Bij de eerdere declaratie van
      ....                      -- INIT behorende body.
   end INIT;

begin
   B := GET_A(10); -- aanroep van bekende GET_A
   ....
end;
```

5.b. Soorten deelprogramma's

De soorten subprograms in Ada zijn net als in Pascal de procedure en de function, maar in Ada kunnen nog als bijzonder geval van functions operatoren gedefinieerd worden.
Procedures in Ada verschillen nauwelijks van wat in Pascal gedaan kan worden (behalve wat de mogelijkheden van parameters betreft) en ze worden geactiveerd door procedure-aanroepen als statement.
Functions mogen in Ada alleen in-parameters hebben (zie onder), d.w.z. dat er waarden via parameters het subprogram in kunnen gaan, maar er kunnen geen nieuwe waarden via parameters uit komen. Het resultaat van wat de functie aflevert op de plaats waar hij is aangeroepen (als expressie of als term of factor van een expressie) mag van elk (ook zelfgemaakt) type zijn, nl. van het (sub)type dat in de specificatie na return genoemd wordt.
Operatoren kunnen een (andere of extra) betekenis krijgen door een nieuwe definitie te geven, voor een willekeurige combinatie van (sub)types van linker- en rechter-operand en van het resultaat. Als de operator voor een zekere combinatie van operanden en resultaat al bestond (bijv. "+" voor INTEGER linker-, rechter-operand en resultaat) dan wordt de operator ge-herdeclareerd (voor de duur van het geldigheidsgebied van deze declaratie), anders is er een nieuwe betekenis bijgekomen: beide gevallen heten overloading (bij herdeclaratie is de oorspronkelijke betekenis wel verborgen, maar hij bestaat nog wel en is

met wat moeite ook te verkrijgen). Dit 'overloaden' kan voor alle soorten subprograms (en een nieuwe betekenis verschilt al van de oorspronkelijke als alleen maar de formele naam van een parameter verschillend is).
Voor operatoren geldt: de prioriteit in een expressie kan niet veranderd worden, een als monadische operator aanwezige operator kan als monadische operator hergebruikt worden, een dyadische operator als dyadische, maar een operator die niet monadisch bestaat, bijv. "*", kan niet monadisch gedefinieerd worden, en alleen voor de volgende operatoren kan een nieuwe definitie gegeven worden:

1. ** abs not
2. * / mod rem
3. + - &
4. < <= > >=
5. and or xor

terwijl voor "=" alleen in een bijzondere, niet-behandelde situatie een nieuwe definitie gegeven kan worden.
Een voorbeeld van twee nieuwe declaraties voor "*" om ook expressies van gemengd type te kunnen maken, is:

```
function "*" (A : INTEGER; B : FLOAT) return FLOAT is
-- Aanroep: INTEGER_OPERAND * FLOAT_OPERAND, het type van het
-- resultaat is: FLOAT.
begin
   return FLOAT(A) * B; -- gebruikt de predefined * voor 2 floats.
end "*";

function "*" (A : FLOAT; B : INTEGER) return FLOAT is
-- Aanroep: FLOAT_OPERAND * INTEGER_OPERAND, het type van het
-- resultaat is: FLOAT.
begin
   return B * A; -- gebruikt de vorige definitie.
end "*";
```

Uit de voorbeelden blijkt al, dat de naam van een gedefinieerde operator bestaat uit de tekens van de operator tussen aanhalingstekens (stringquotes). Als operator gebruikt worden de stringquotes weggelaten. Een operator kan in aanroep ook als functie gebruikt worden:

```
C := "*" (INT_A, FLOAT_B); -- doet het zelfde als
C := INT_A * FLOAT_B;
```

Dit heeft pas zin als een dubbelzinnigheid als gevolg van te veel 'overloading' opgelost moet worden, bijv. om de operator "*" uit het pakket MIXED_OPS aan te wijzen, nl. door: MIXED_OPS."*" .

5.c. Externe specificatie

Een subprogram declaration bestaat uit een (externe) subprogram specificatie gevolgd door een ;. Ook een subprogram body begint met een subprogram specificatie, maar daarna volgt het symbool is en het interne deel van de body (zie 5.d).
Zo'n externe specificatie definieert:

- de naam waaronder het deelprogramma aangeroepen kan worden,
- de namen van eventuele formele parameters, hun type en wat er mee gedaan kan worden, mogelijk ook een default-expressie ervoor,

voor functies (en operatoren) het (sub)type van hun resultaat.

De algemene vorm is: voor een procedure:
procedure PROCEDURE_NAAM <formeel deel>
voor een function of operator:
function FUNCTIE_NAAM <formeel deel> return <(sub)type naam>
waarin FUNCTIE_NAAM ook een operator-string ("+") kan zijn.

Het formele deel kan afwezig zijn, wat dan een procedure of functie zonder parameters definieert (voor operatoren niet toegestaan, zie vorige paragraaf). Als het aanwezig is heeft het de vorm:

```
( <parameter specificatie>
  ; <parameter specificatie>
  enz.
)
```

d.w.z. tussen haakjes een aantal parameter-specificaties, gescheiden door puntkomma's.

Een parameter-specificatie heeft op zijn beurt de vorm:
<identifier-lijst> : <mode> <(sub)type-naam>
in 1 geval (nl. als de mode in is) mogelijk nog een default-expressie voor de in de identifier-lijst genoemde formele parameters bevattend:
<identifier-lijst> : in <(sub)type-naam> := <default expr.>
Afhankelijk van de mode zijn er 3 soorten parameters van subprograms (bij meerdere identifiers in 1 parameter-specificatie geldt voor alle formele parameters dezelfde mode en (sub)type):

1) mode in:
   Als een formele parameter de mode in heeft moet de actuele parameter bij aanroep een expressie zijn van het genoemde (sub)type:
   Bijv.:
   ```
   X0, X1, X2 : in FLOAT
   AANTAL : POSITIVE := 100
   ```
   Het 'reserved word' in mag worden weggelaten, wat we bij procedures meestal niet, bij functions meestal wel doen, omdat de 2-e en 3-e vorm bij functions niet toegestaan zijn. In specificaties voor parameters van een operator mogen geen default-expressies staan. Zie hoofdstuk 5.e voor rol van de default-expressie.
2) mode out:
   Voor een out parameter (alleen voor procedures) geldt, dat de actuele parameter bij aanroep een variabel object moet zijn van het genoemde (sub)type. Het subprogram mag een nieuwe waarde toekennen aan de parameter. Bijv.:
   ```
   X : out A
   ```
3) mode in out:
   Voor een in out parameter (alleen voor procedures) geldt ook, dat actuele parameter bij aanroep een variabel object moet zijn van het genoemde (sub)type. Het subprogram mag de aanwezige waarde van de parameter lezen en er een (nieuwe) waarde aan toekennen. Bijv.:
   ```
   X : in out VECTOR
   ```

M.a.w. bij 'mode' out en 'mode' in out en bij specificatie van operatoren mogen geen 'default-expressies' gegeven worden, en functies (waaronder operatoren) mogen alleen in-parameters hebben, waarbij het symbool in mag worden weggelaten.

Voor functions tenslotte geeft de (sub)type-naam na het 'reserved word' return het (sub)type aan van het resultaat dat een aanroep aflevert. Bijv. (in de vorm van subprogram declaraties):

```
procedure INIT( X : out A ) -- de ene parameter heet X, een
-- actuele parameter bij aanroep moet een variabele van (sub)type A
-- zijn, maar INIT mag een waarde die X vooraf had niet lezen.
procedure NEW_LINE; -- (deze) NEW_LINE heeft geen parameters.
function N_FACTORIAL(N : INTEGER) return FLOAT;
procedure FIND(X0 : in REAL := 0.0; X : in REAL_ARRAY;
               INDEX : out INTEGER) -- levert in INDEX de index
-- van de plaats in array X waar de waarde X0 staat.
-- Voor X0 is een default-expressie gegeven.
```

In het laatste voorbeeld zal men vaak uit vrees dat het array X gekopieerd wordt de zogenaamde 'mode' van X liever in out maken, maar het is in Ada niet zeker dat er dan niet gekopieerd wordt (misschien zelfs twee keer, zie laatste alinea van deze paragraaf).

Doordat een parameter en een function-resultaat van elke (sub)type mogen zijn, zijn de mogelijkheden in Ada veel groter dan in Pascal, want het type kan in Ada 'unconstrained' zijn, waarna een procedure kan worden aangeroepen met arrays van verschillende lengte.
Wat in Ada niet kan is aan een procedure of functie als parameter een procedure- of functie-naam meegeven.
Het belangrijkste verschil is verder het parameter-mechanisme: bij Pascal is dit 'call-by-value' en 'call-by-reference' voor resp. value- en var-parameters, Ada geeft met de 'mode' alleen maar aan wat er een subprogram in en/of uit kan gaan via de parameters, d.w.z. waarden van een (sub)type, maar Ada schrijft met name voor samengestelde types niet voor hoe dat tot stand komt. Berekende waarden voor out en in out parameters mogen tijdens uitvoering van de body-statements in onzichtbare lokale variabelen bewaard worden, terwijl de meegegeven variabelen (de actuele parameters) pas bij verlating van de body worden bijgewerkt.

5.d. Interne specificatie

De interne specificatie bestaat net als voor Pascal uit een mogelijk leeg declaratie-deel, gevolgd door het opdrachten-deel. Een 'subprogram body' heeft daarom de volgende vorm:

```
<subprogram specificatie>                -- zie 4.c.
is
   <declaratie-deel>
begin
   <rij statements>
   -- eventueel gevolgd door exception handlers.
end <subprogram identifier>;
```

De subprogram identifier na het end van de body mag wel weggelaten worden, maar er mag niet iets anders staan. Dit kan een extra veiligheid zijn als een programma geschreven wordt.

Van het declaratie-deel is inmiddels wel bekend dat daar alle mogelijke lokale declaraties gegeven kunnen worden, waarbij ook namen die buiten

het subprogram bestaan een lokale nieuwe betekenis kunnen krijgen. Verder kunnen in het 'tweede' deel van het declaratie-deel ook bodies van lokale subprograms gegeven worden.

De rij statements definieert de werking van een subprogram als dat door een aanroep geactiveerd wordt. Naast de al bekende statements moet hier nu de 'return statement' genoemd worden, die ook midden in het statement-deel terugkeer naar de plaats van aanroep van het subprogram veroorzaakt. Voor procedures is de vorm:

```
return;
```

en deze kan bijvoorbeeld in een 'if-statement' voorkomen. Als een procedure geen return bevat, vindt terugkeer plaats als de rij statements afgelopen is.
Voor functies (en dus ook operatoren) is de return-statement van de vorm

```
return <expressie>;
```

verplicht, omdat daarmee tevens wordt aangegeven welke waarde de functie of operator aflevert. De expressie moet een waarde zijn van het in de specificatie gegeven resultaat-type, of een (constrained) subtype daarvan. Een functie mag alleen met een return-statement verlaten worden, maar subprograms mogen wel meerdere return-statements bevatten. Bijv.:

```
function NFAC(N : INTEGER) return FLOAT is
   X : FLOAT := 1.0;
begin
   if N < 0 then return 0.0; end if;
   for I in 2 .. N loop
      X := X * FLOAT(I); -- met type-conversie
   end loop;
   return X;
end NFAC;
```

Binnen de body van een subprogram kan gebruik gemaakt worden van:
- lokaal gedeclareerde namen,
- de formele parameters,
- alle namen waarvoor geldt dat dit subprogram in het geldigheidsgebied ervan staat, i.h.b. de subprogram-naam zelf.

Voor globale dingen, die verborgen worden door een lokale herdeclaratie met dezelfde naam, geldt dat ze met een uitgebreide naam (zoals BUITEN.X in hoofdstuk 2.e) nog wel bereikbaar zijn.
Recursieve deelprogramma's hebben dezelfde betekenis als in Pascal.

5.e. Aanroep en parameter-associatie

Aanroep van subprograms (voor procedures als statement, voor functies als 'primary' in een expressie, voor operatoren als operator in een formule) heeft voor operatoren de traditionele betekenis (associatie van linker- en rechter-operand met de eerste en tweede formele parameter, voor monadische operatoren de rechter-operand met de enige parameter). Voor andere subprograms is de vorm van de aanroep:

```
<subprogram naam> <eventueel actuele-parameter deel>
```

en een actuele-parameter deel heeft de vorm:

```
( <parameter associatie>
  , <parameter associatie>
  enz.
)
```

m.a.w. tussen ronde haakjes een of meer parameter-associaties, gescheiden door komma's.

Een parameter-associatie kan zijn:
1) een positionele associatie: een expressie of variabele van het (sub)type van de met de positie corresponderende formele parameter, een expressie als de formele parameter van de 'mode' in is, een variabele als de mode out of in out is.
2) een benoemde associatie ('named association'): de vorm is:
   <formele naam> => <expressie, resp. variabele>
   De parameter-naam geeft aan, met welke formele parameter de expressie of variabele geassocieerd moet worden.

Als voor een formele parameter de associatie ontbreekt moet er een default-expressie voor gegeven zijn in de parameter-specificaties (dus alleen voor 'mode' in parameters) en bij aanroep wordt dan de waarde van die expressie genomen.
Zodra een benoemde associatie wordt gebruikt in een aanroep, moet de associatie van de volgende formele parameters van het subprogram ook benoemd of afwezig zijn.

Als een variabele wordt verwacht, mag de actuele parameter ook een type-conversie van een variabele zijn.

Voorbeelden:

```
INIT(B);        -- of:
INIT(X => B);   -- zie 5.a en 5.c.

FIND(1.0, VEC1, J);      -- zie 5.c.
FIND(X => VEC1,
     INDEX => J,
     X0 => PI);
FIND(INDEX => J, X => VEC1); -- met default voor X0
```

Pascal kent de mogelijkheid van 'named' parameter-associatie niet, evenmin als de vrijheid om bij in parameters, waarvoor een default-expressie gegeven is, de associatie helemaal weg te laten. Als de positionele associatie gebruikt wordt in Ada, lijken de aanroepen van procedures en functies wel veel op de aanroepen in Pascal, hoewel de echte uitvoering kan verschillen door een ander parameter-mechanisme maar dit zal bij 'net' gebruik niet tot andere uitkomsten leiden. Tenslotte kent Pascal geen 'overloading', daar een herdeclaratie altijd een vorige declaratie volledig verbergt.

6. I/O

Pascal kent principiëel alleen sequential files, en in de taal zijn een aantal standaard-namen beschikbaar om met files te werken. Naar structuur zijn er eigenlijk twee soorten file, nl.

- files waarop (sequentieel) waarden van een bepaald basistype kunnen staan, waarbij lezen met een volgende opzet kan geschieden:

```
reset(filevar);
WHILE NOT eof(filevar) DO
   read(filevar, basistypevar);
```

  terwijl schrijven in de volgende opzet kan plaatsvinden:

```
rewrite(filevar);
WHILE NOT voldoendeuitvoer DO
   write(filevar, basistypewaarde);
```

- tekst-files, d.w.z. files bestaande uit regels, waarbij de regels bestaan uit symbolen: voor lezen kunnen reset, eof, read en readln gebruikt worden, voor schrijven: rewrite, write en writeln. Bij read kan de variabele behalve van het type char ook van een getal-type zijn en dan worden voldoende symbolen gelezen die een getal-notatie voorstellen (zonder format), bij write kunnen de waarden van de types: char, string, boolean, integer en real zijn en er vindt conversie plaats naar een symbolen-rij.

Andere soorten files ('random access', of met anders gekodeerde inhoud) behoren in Pascal niet tot de standaard.

In Ada kan met 'files' gewerkt worden door types en subprograms te gebruiken uit bibliotheek-pakketten voor sequentieel of met random-access lezen en schrijven, en voor in-/uitvoer van teksten. Daarbij blijft een bepaalde fysieke structuur van files volledig verborgen, alle gebruik geschiedt via de bijgeleverde subprograms, zoals: OPEN, READ, WRITE, END_OF_FILE-test.
Zonder in te gaan op een aantal nieuwe uitdrukkingsmogelijkheden van Ada, die hierbij een rol spelen (nl. private types, exceptions, packages, context clauses, en generic instantiatie), volgen hier enkele voorbeelden van gebruik van bibliotheek-pakketten, die voor eenvoudige invoer en uitvoer standaard aanwezig zijn in Ada.
De voorbeelden behandelen: het verkrijgen en gebruik van files voor een vast elementen-type (in Pascal: FILE OF basetype) voor resp. sequentiëel en direct access, het werken met tekst-files.

Algemeen eerst: een file kan worden geopend (of gecreëerd) met een procedure-aanroep, waarbij een string-waarde wordt meegegeven, die de implementatie kan verbinden met een externe file. Verder heeft elke file een file-mode: IN_FILE of OUT_FILE al naar gelang er op een bepaald moment mee gelezen of geschreven mag worden, en de file-mode kan (als daar permissie voor is) door een RESET-aanroep worden gewijzigd, terwijl er voor direct-access files ook nog een waarde INOUT_FILE is.

6.a. Gebruik van sequentiele files van een basistype

Het 'generic package' SEQUENTIAL_IO kan gebruikt worden om op files waarden van een BASIS_TYPE te schrijven, resp. om zulke waarden weer te lezen. Een compleet voorbeeld is:

```
with SEQUENTIAL_IO; -- Een context clause.
procedure MAIN is
   subtype INT is INTEGER range 0 .. 1000;

   package ABOUT_INT_FILE is
      new SEQUENTIAL_IO( ELEMENT_TYPE => INT );
   -- Maakt een kopie-package voor het element-(sub)type INT.
   use ABOUT_INT_FILE;
   -- Nu zijn een file-type en de lees- en schrijfprocedures
   -- in het package direct zichtbaar.

   KWADR_FILE : FILE_TYPE;
   -- Bij meerdere element-types zou ik
   -- KWADR_FILE : ABOUT_INT_FILE.FILE_TYPE;
   -- geschreven hebben.

   TERM : INT;
   SUM : INTEGER := 0;

begin
   CREATE(KWADR_FILE, OUT_FILE, "kwadrtn");
      -- associatie met externe file 'kwadrtn'.
   for I in 1 .. 31 loop
      WRITE(KWADR_FILE, I * I);
   end loop;

   -- Nu file herpositioneren op het begin voor lezen:
   RESET(KWADR_FILE, MODE => IN_FILE);

   while not END_OF_FILE(KWADR_FILE) loop
      READ(KWADR_FILE, TERM);
      SUM := SUM + TERM;
   end loop;
end MAIN;
```

Nadat (voor elk gewenst 'element-type') een zgn. instantiatie van het 'generic package' is gemaakt, zijn er net als in Pascal beschikbaar een file-type FILE_TYPE, subprograms als READ, WRITE en END_OF_FILE, en bovendien:

CREATE en OPEN (voor het associeren van een interne file met een externe file die extern een bepaalde naam heeft),
DELETE en CLOSE (voor het verwijderen of sluiten van een externe file),
RESET(file-object, MODE => IN_FILE); en
RESET(file-object, MODE => OUT_FILE); voor:
resp. de reset en rewrite van Pascal,
en functies MODE, NAME, FORM en IS_OPEN om de betreffende waarden van een file te verkrijgen.

We merken nog op dat bij meerdere file-types naast elkaar we als prefix van de identifier FILE_TYPE de package-naam moeten vermelden om aan te geven uit welk package we het file-type gebruiken. Voor de procedures zoals READ en WRITE hoeft dit dan nog niet, omdat die 'overloading' wel opgelost kan worden.

6.b. Direct access files

Voor files van een enkel element-type kan ook met direct access gewerkt worden, maar dan moet men het 'generic package' DIRECT_IO gebruiken. De file-mode kan dan ook INOUT_FILE zijn, en er zijn subprograms SET_INDEX, INDEX en SIZE, terwijl READ en WRITE parameters FROM, resp. TO hebben om een gewenste plaats te gebruiken.

6.c. Tekst files

Voor het invoeren en uitvoeren van teksten is een bibliotheek-pakket TEXT_IO beschikbaar, dat de types FILE_TYPE en FILE_MODE bevat, verder de subprograms CREATE, OPEN, CLOSE, DELETE, RESET, MODE, NAME, FORM, IS_OPEN en END_OF_FILE als tevoren, procedures SET_INPUT en SET_OUTPUT om defaults voor de invoer- en uitvoer-files te geven, en daarnaast subprograms om net als in Pascal regels te skippen of af te sluiten, GET en PUT voor CHARACTERs, GET en GET_LINE, PUT en PUT_LINE voor STRINGs. Ook subprograms om pagina's te skippen of af te sluiten, regellengte of paginalengte te definieren, en naar een positie op de regel of naar een regel van de pagina te skippen. Voor details wordt verwezen naar de specificatie van TEXT_IO, Ada Reference Manual, sectie 14.3.10.

Om waarden van INTEGER of FLOAT type, of van zelfgemaakte scalar, integer of real types te verwerken, na conversie van of naar een symbolenrij, moet voor elk type eerst een daarvoor bestemd generic (deel)package uit TEXT_IO geinstantieerd worden, nl. INTEGER_IO, FLOAT_IO, FIXED_IO of ENUMERATION_IO. Dan zijn voor het meegegeven type subprograms beschikbaar als GET, GET_LINE, PUT en PUT_LINE met extra parameters als:

WIDTH voor de lengte van de te converteren of geconverteerde symbolenrij,

BASE voor de basis van het talstelsel (bij PUT), (men kan ook de waarde van de variabele DEFAULT_BASE van het subtype INTEGER range 2 .. 16 uit TEXT_IO veranderen)

FORE, AFT en EXP (bij PUT): voor reële waarden het aantal posities voor en na de decimale punt en na de E van de exponent.

GET en PUT zijn ook beschikbaar met een formele parameter FROM, resp. TO van type STRING, en dan wordt niet met een file gewerkt maar vindt er conversie plaats van een string-waarde naar een getal-variabele, resp. van een getal-waarde naar een string-variabele.
Bij gebruik van ENUMERATION_IO zijn de mogelijkheden analoog, maar dan moet de gelezen string een correcte enumeration-waarde voorstellen, dus een identifier of character-literal van het goede type.

Hieronder een voorbeeld voor de invoer van een aantal gehele getallen, en wat uitvoer van tekst met getallen ertussen.

```
with TEXT_IO; use TEXT_IO; -- Een context clause,
      -- dit maakt de onderdelen van TEXT_IO zichtbaar.
procedure MAIN is
   subtype INT is INTEGER range 0 .. 1000;

   package MY_INT_IO is
      new INTEGER_IO( NUM => INT );
   use MY_INT_IO;
```

```
      -- Nu zijn voor INT de lees- en schrijfprocedures
      -- beschikbaar (met conversie van/naar symbolenrij).

      DATA_FILE, UITVOER : FILE_TYPE;
      -- Dit is de FILE_TYPE uit TEXT_IO.

      TERM, TELLER : INT;
      SOM : INTEGER := 0;

   begin
      OPEN(DATA_FILE, IN_FILE, "kwadrtn"); SET_INPUT(DATA_FILE);
      -- DATA_FILE is een bestaande tekst-file, met externe naam
      -- 'kwadrtn', en ik hoef in GET-statements niet meer de
      -- file (DATA_FILE) te vermelden.
      -- N.B. dit is niet de uitvoerfile van het voorbeeld in
      -- paragraaf 6.a., want dat was een gekodeerde file. Ik neem
      -- wel aan dat er nu (in tekst) 31 getallen op staan.

      TELLER := 0;
      begin
         loop
            GET(TERM); -- Lees formatloos, en neem aan dat
               -- END_OF_FILE-test net als in Pascal niet
               -- bruikbaar is, dus vang de exception op.
            TELLER := TELLER + 1; SOM := SOM + TERM;
         end loop;
      exception
         when END_ERROR => null; -- opgevangen.
            -- een DATA_ERROR wordt niet opgevangen.
      end;

      OPEN(UITVOER, OUT_FILE, "output"); SET_OUTPUT(UITVOER);
         -- associatie met externe file 'output'.
      PUT_LINE("Lees een tekstfile met getallen:");
      PUT("de som van de ");
      PUT(ITEM => TELLER, WIDTH => 4);
      PUT(" getallen bedraagt ");
      PUT(SOM, WIDTH => 8);
      NEW_LINE;
   end MAIN;
```

De externe file 'output' krijgt de waarde:

```
Lees een tekstfile met getallen:
de som van de   31 getallen bedraagt     10416
```

7. CONCLUSIES

Aangetoond is dat er voor de hogere programmeertalen Pascal en Ada een aanzienlijke overlapping bestaat, zowel wat betreft de uitdrukkingsmogelijkheden van de talen als wat betreft de notaties van de overlappende delen. Daarbij is gebleken dat op enkele uitzonderingen na de uitdrukkingsmogelijkheden van Pascal door die van Ada overdekt worden. Die uitzonderingen betreffen het 'set-type' van Pascal (al is dit wel met een data-structuur als een array van BOOLEANs, en een uitgebreid pakket van manipulatie-functies in Ada te simuleren) en het ontbreken in Ada van procedurele parameters van subprograms (waarvoor het 'generics'-gereedschap niet voldoende vervanging biedt).
Daarnaast biedt Ada voor het overlappende deel van de talen vaak meer mogelijkheden, zoals: aggregates, initialisaties in de declaraties, operatie-declaraties, gevarieerde associatie van formele en actuele parameter in aanroepen, en een groter repertoire van lees- en schrijf-procedures.
Minder goed op zijn waarde te schatten is het verschil in de compatibiliteitsregels bij het maken van nieuwe types: voor Pascal geldt 'declaratie equivalentie' voor compatibiliteit, in Ada geldt 'strong typing'. Dit betekent in Ada, dat twee types verschillend zijn als ze te herleiden zijn tot verschillende type-definities, wat ook 'derived types' betreft (zie 4.a). In Pascal is dit (officieel) ook zo maar daar zijn de vormen TYPE newtype = typenaam; en
TYPE subrtype = subrange; eigenlijk subtype-definities, en types die zo gemaakt worden zijn compatibel met het basis-type. Het Ada-concept helpt de programmeur door bij fouten te waarschuwen, maar de verbodsregels worden al gauw als hinderlijk ervaren.
Tenslotte is er in Ada een groot aantal niet behandelde gereedschappen waardoor andere zaken in programmatuur uit te drukken zijn, met name: fouten-opvang, bibliotheek-opbouw, real-time programmeren, en parametriseren met types en bijbehorende operaties van die types.

Als uitdrukkingsmiddel biedt Ada veel meer mogelijkheden dan Pascal. Dat de taal daardoor moeilijker is ligt voor de hand, en niet voor elk klein probleem zal men derhalve liever Ada dan Pascal gebruiken. Weinig valt thans te zeggen van Ada betreffende de snelheid van verwerking van vertaalde programma's, zoals bekend een van de sterke punten van vele implementaties van Pascal. Het ligt voor de hand dat althans het vertalen in Ada merkbaar kostbaarder zal zijn dan in Pascal.

Geconcludeerd mag worden, dat een Pascal-programmeur al een flink deel van Ada beheersen kan door kennis te nemen van de hiervoor behandelde overeenkomsten. Daarbij moet wel altijd ervoor gewaarschuwd worden, dat een goede stijl van programmeren in de ene taal zeker niet een goede stijl voor het gebruik van de andere taal is, als in de andere taal ook nog nieuwe uitdrukkingsmiddelen beschikbaar zijn. Bij een overgang van Pascal naar Ada geldt dit evengoed.

LITERATUUR

Reference Manual for the Ada Programming Language, ANSI MIL-STD 1815 A, US Department of Defense (AJPO), 1983.

Jensen, K. & N. Wirth: Pascal, User Manual and Report, Springer Verlag, Berlijn, 1974.