Gliederung der Grundvorlesung

1. Grundlagen der Programmierung

~~1.1 Algorithmen und Sprachen~~
~~1.2 Aussagen über Algorithmen~~
~~1.3 Daten und ihre Strukturierung~~
1.4 / 1.5 Grundbegriffe der Programmierung und die Sprache Ada 95
1.6 Semantik von Programmen
1.7 Komplexität von Algorithmen und Programmen

Gliederung des Kapitels

1.4/1.5 Programmierung (und die Sprache Ada 95)

1.4.1 Blöcke, Deklarationen und Ausnahmen
1.4.2 Prozeduren und Funktionen
1.4.3 Moduln
1.4.4 Generizität
1.4.5 Vererbung
1.4.6 Abstrakte und konkrete Datentypen
1.4.7 Objekte
1.4.8 Grundprinzipien, Paradigmen der Programmierung

1.4.1 Blöcke, Deklarationen und Ausnahmen

Ein Name ist in der Regel nur in einer bestimmten Umgebung eindeutig. Zum Beispiel wird es in einer Vorlesung mit 200 Hörer(inne)n meist nur höchstens einen "Paul Müller" oder eine "Rita Weber" geben, bundesweit dagegen können viele Personen mit diesen Namen besitzen.

In einem Programm muss jeder Name eine eindeutige Bedeutung besitzen. Fügt man jedoch zwei Programme zu einem neuen zusammen, so können hierbei zwei gleiche Namen, die bisher in den beiden getrennten Programmen vorkamen, zusammentreffen und zu einer Mehrdeutigkeit führen. Man wird daher jedem Namen eine "Umgebung" zuordnen, in der er gültig ist. Diese Umgebung, also ein in sich geschlossenes Programmstück mit Deklarationen, nennt man einen "Block".

1.4.1.1: *Definition und Ada-Syntax für Blöcke in Ada*
(siehe 1.1.6.6, "block_statement" steht für "Block")

Ein Block ist eine in sich geschlossene, durch begin ... end geklammerte Folge von Anweisungen, die einen Deklarationsteil am Anfang und Ausnahmebehandlungen (um Fehlersituationen abzufangen) am Ende besitzen kann.

Die im Deklarationsteil vereinbarten Namen können nur innerhalb dieses Blocks und seiner Unterblöcke verwendet werden. Wird der Block verlassen, so sind diese Namen undefiniert oder "unbekannt".

Wird in einem Unter-Block eine Variable, die in einem Ober-Block deklariert wurde, neu deklariert, so wird der äußere Name "ausgeblendet" und im Unter-Block kann nur die dort deklarierte Variable unter diesem Namen angesprochen werden.

Ada-Syntax ("block_statement" steht für "Block")

```
block_statement ::=
        [block_statement_identifier ":"]
        ['declare'  declarative_part]
        'begin'  handled_sequence_of_statements
        'end' [block_identifier] ";"

handled_sequence_of_statements ::=
        sequence_of_statements
        ['exception' exception_handler {exception_handler}]

sequence_of_statements ::= statement {statement}
```


Beispiel 1.4.1.2: Blöcke und Blockschachtelung

```
procedure BSP is
K: integer;                                   Rumpf der procedure
begin K := 7; ...                             = Block 0
...
   declare K, L: float;                                  Block 1
   begin
   K := 0.7; L := K+2.0;
   ...
      declare K,M: char;                                 Block 2
      begin
      K := 'A'; M := Succ(K); L := L+1.0; ...
      ...
      end;
   end;
   ...
end BSP;
```

Beispiel: Namenszuordnung

Es ist stets das K gemeint, das im kleinsten umfassenden Block deklariert ist.

```
procedure BSP is
K: integer;                                          Rumpf der procedure
begin K := 7; ...
...
    declare K, L: float;                                        Block 1
    begin
    K := 0.7; L := K+2.0;
    ...
        declare K, M: char;                                     Block 2
        begin
        K := 'A'; M := Succ(K); L := L+1.0; ...
        ...
        end;
    end;
    ...
end BSP;
```


Beispiel: Namenszuordnung

```
procedure BSP is
K: integer;                                          Rumpf der procedure
begin K := 7; ...
...
    declare K, L: float;                                        Block 1
    begin
    K := 0.7; L := K+2.0;
    ...
        declare K, M: char;                                     Block 2
        begin
        K := 'A'; M := Succ(K); L := L+1.0; ...
        ...
        end;
    end;
    ...
end BSP;
```


Festlegung 1.4.1.3:
Steht ein Name X in einem Programm mit Blöcken an einer Stelle s, so bezieht sich X stets auf die Deklaration von X, die sich am Anfang des kleinsten, s umfassenden Blocks (oder Deklarationsteils einer Prozedur) befindet.

Hinweis (hier für Sie noch nicht zu verstehen):
Diese Festlegung gilt auch für die Benutzung von Namen in Prozeduren und Moduln *(nach Anwendung der Kopierregel, siehe später).*
Allerdings darf sich ein Name an der Stelle s nie auf eine Deklaration beziehen, die die Stelle s nicht "statisch" (also textuell) umfasst. Man denke z.B. an gleichrangige nebeneinander stehende Prozeduren, die sich gegenseitig aufrufen und gleiche Namen für ihre Variablen enthalten.

1.4.1.4: Speicherverwaltung mit Blöcken

Die Variablen, die in Blöcken vereinbart werden, werden grundsätzlich im Kellerspeicher abgelegt. Die Anlegung der Speicherbereiche erfolgt hierbei während der Laufzeit: Sobald der Block betreten wird, wird der Platz für die neu vereinbarten Variablen reserviert ("Allokation" von Speicher).

Wird der Block verlassen, so wird der Speicherplatz wieder frei gegeben.

Wir erläutern dies an folgendem Beispiel.

Programmzeiger

Beispiel

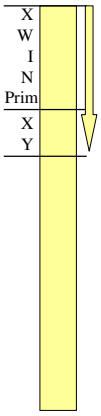
```
procedure BSP2 is    -- nicht auf den Inhalt achten!
X, W: float; I, N: integer; Prim: Boolean;
begin get(N);
   declare X, Y: float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```

Lokaler Keller-Speicher

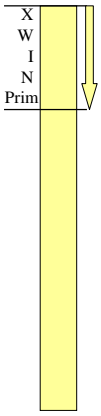

```
procedure BSP2 is    -- nicht auf den Inhalt achten!
X, W: float; I, N: integer; Prim: Boolean;
begin get(N);
   declare X, Y: float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```

X
W
I
N
Prim

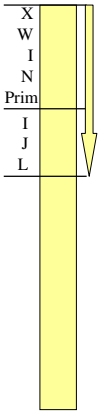
```
procedure BSP2 is    -- nicht auf den Inhalt achten!
X, W: float; I, N: integer; Prim: Boolean;
begin get(N);
   declare X, Y: float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```



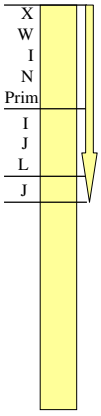

```
procedure BSP2 is    -- nicht auf den Inhalt achten!
X, W: float; I, N: integer; Prim: Boolean;
begin get(N);
   declare X, Y: float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```



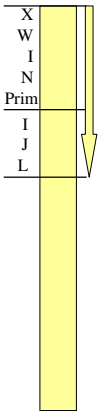

```
procedure BSP2 is    -- nicht auf den Inhalt achten!
X, W: float; I, N: integer; Prim: Boolean;
begin get(N);
   declare X, Y: float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```



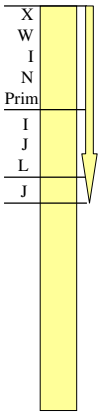

```
procedure BSP2 is    -- nicht auf den Inhalt achten!
X, W: float; I, N: integer; Prim: Boolean;
begin get(N);
   declare X, Y: float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```



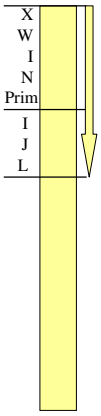

```
procedure BSP2 is    -- nicht auf den Inhalt achten!
X, W: float; I, N: integer; Prim: Boolean;
begin get(N);
   declare X, Y: float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```



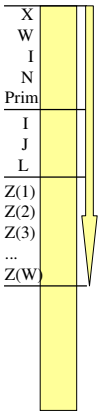

```
procedure BSP2 is    -- nicht auf den Inhalt achten!
X, W: float; I, N: integer; Prim: Boolean;
begin get(N);
   declare X, Y: float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```



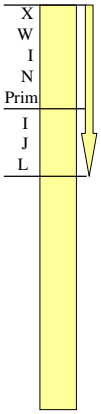
```
procedure BSP2 is    -- nicht auf den Inhalt achten!
X, W: float; I, N: integer; Prim: Boolean;
begin get(N);
   declare X, Y: float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```



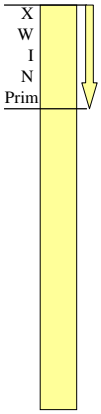

```
procedure BSP2 is    -- nicht auf den Inhalt achten!
X, W: float; I, N: integer; Prim: Boolean;
begin get(N);
   declare X, Y: float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```

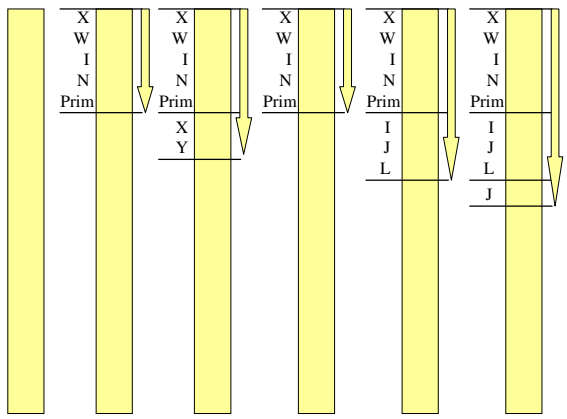


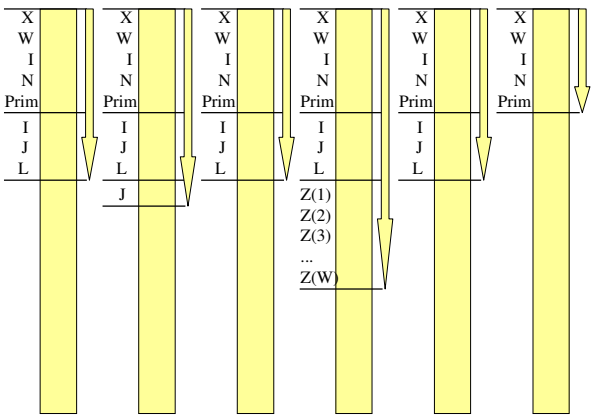

```
procedure BSP2 is    -- nicht auf den Inhalt achten!
X, W: float; I, N: integer; Prim: Boolean;
begin get(N);
   declare X, Y: float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```

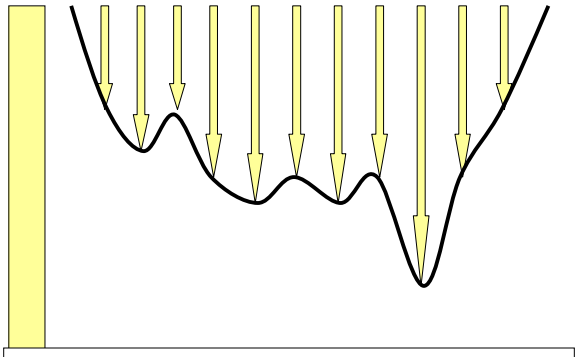



```
procedure BSP2 is    -- nicht auf den Inhalt achten!
X, W: float; I, N: integer; Prim: Boolean;
begin get(N);
   declare X, Y: float;
   begin X := Float (N);
        while ....; -- berechne angenähert die Wurzel Y aus N
        W := Integer (Y);
   end;
   declare I, J, L: integer;
   begin I := 2;
        while (I <= W) and (I*I /= N) loop
             declare J: integer;
             begin I:=I+1; J:=W-I; if J*J=N then ... end if;
             end;
        end loop;
        Prim := not (I*I = N);
        declare Z: array (1..W) of integer;
        begin   ...
        end;
   end;
   I := N; while I>0 loop I:=I-1; ... end loop; ....
end BSP;
```









Der lokale Speicher ist ein pulsierender Keller-Speicher.
Er lässt sich durch verschachtelte Blöcke steuern.

1.4.1.5: Vorteile von Blöcken

1. Sie bilden kleinste in sich geschlossene Programmstücke, die man getrennt entwickeln, verstehen und optimieren kann.

2. Die benötigten Hilfsvariablen und Zwischenrechnungen sind nach Abarbeitung des Blocks verschwunden.

3. Aus Blöcken lassen sich größere Programmeinheiten zusammenstellen, ohne dass Namenskonflikte auftreten.

4. Mit Blöcken kann man Einfluss auf den kellerartigen Speicherplatz, der keiner Speicherbereinigung unterliegt, nehmen und den Speicherplatz gezielt selbst verwalten.

Hinweis: Diese Vorteile lassen sich auch mit Prozeduren und Paketen erreichen; allerdings entsteht bei ihnen ein zusätzlicher Verwaltungsaufwand bei der Programmausführung.

1.4.1.6: Mit den Blöcken sind die Begriffe Gültigkeitsbereich (oder Sichtbarkeitsbereich) und Lebensdauer von Namen verbunden.

Die Lebensdauer eines Namens ist der Block, in dem der Name deklariert wurde, und zwar genau ab der Stelle, an der der Name deklariert wird. Man sagt, der Name *lebt* ab dieser deklarierenden Stelle (im Falle einer Variablen bedeutet dies: Die Variable besitzt nun einen Speicherplatzbereich im lokalen Kellerspeicher), aber nur innerhalb dieses Blockes; er "stirbt" oder ist unbekannt, sobald dieser Block verlassen wird.

Spezialfall in Ada: Die Lebensdauer einer *Laufvariablen* ist die jeweilige Schleife. Außerhalb der Schleife ist die Laufvariable nicht existent.

Der Gültigkeitsbereich oder Sichtbarkeitsbereich eines Namens ist der Teil der Lebensdauer, in dem auf das Objekt, das mit diesem Namen verbunden ist, unmittelbar über den Namen zugegriffen werden kann. Nicht gültig (oder sichtbar, engl. *visible*) ist der Name in allen Unterblöcken, in denen er neu deklariert wird. Er "taucht wieder auf" (= er wird wieder sichtbar), sobald diese Unterblöcke verlassen werden.

Beachte: Der deklarierende Block umfasst den gesamten Bereich von declare bis end, und die allgemeine Regel, dass ein Name sichtbar sein muss, wenn man ihn verwenden will, gilt auch für den Deklarationsteil. Daher müssen wir bei access-Datentypen den Typ T, auf den verwiesen wird, zuvor bezeichnen, dann den access-Typ definieren und anschließend T selbst.

Beispiel:

```
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
procedure Laufvar is
J,    L: integer;  K: integer;
begin   J:=1;  L:=1;
        for J in L..4 loop K := J + 3;  end loop;
        L:=J+2;
        put(J); put(L); put(K);
end Laufvar;
```

Implizite Deklaration der Variablen J

Dieses Programm liefert daher die Ausgabewerte 1, 3 und 7.

Hinweis: Die Formulierung for J in J..4 loop ... ist nicht zulässig, da die Variable "J" in "J..4" bereits die Laufvariable J wäre, die aber in dieser Unterbereichsfestlegung nicht benutzt werden darf. In Ada sollte man einer Laufvariablen keinen Namen geben, der hier lebt (Fehlergefahr).

1.4.1.7: In der Definition von Blöcken tritt der Deklarationsteil (declarative_part) auf. Wir betrachten diesen genauer:

declarative_part ::= {declarative_item}

declarative_item ::= basic_declarative_item | body

basic_declarative_item ::=
basic_declaration | representation_clause | use_clause

basic_declaration ::=
type_declaration | subtype_declaration |
object_declaration | number_declaration |
subprogram_declaration | abstract_subprogram_declaration |
package_declaration | renaming_declaration |
exception_declaration | generic_declaration |
generic_instantiation

```
object_declaration ::=
        defining_identifier_list ":" ['aliased'] ['constant']
              subtype_indication [":=" expression] ";"        |
        defining_identifier_list ":" ['aliased'] ['constant']
              array_type_definition [":=" expression] ";"     |
        single_task_declaration  |
        single_protected_declaration

number_declaration ::=
       defining_identifier_list ":" 'constant' ":=" static_expression;
```


1.4.1.8 Konstanten und Initialisierung: Man sieht, dass man

1. Variablen bei der Deklaration initialisieren darf und
2. Konstanten einführen kann, die dann im Programm nicht mehr verändert werden dürfen. (Nur bei numerischen Typen ist es erlaubt, den Datentyp wegzulassen.)

Beispiele. Im Deklarationsteil ist also zugelassen:

```
Pi : constant float := 3.14159265359;
E : constant := 2.718281828459;
K: integer := 24;  EinK: constant := 1024;
X: array (1..3) of float := (1.0, 2.0, 3.0);
Y: array (1..EinK) of Character := (1=>1, 2=>7, others=>0);
type Tag is (Mo, Di, Mi, Do, Fr, Sa, So);
type ST is array (1..5) of Tag;
Ueberschrift: constant ST := (Mo, Di, Mi, Do, Fr);
Mitte_der _Woche: constant Tag := Mi;
```


Fortsetzung zu "declarative_part": (zur Kenntnisnahme)

```
representation_clause ::= attribute_definition_clause   |
          enumeration_representation_clause   |
          record_representation_clause    |    at_clause

use_clause ::= use_package_clause | use_type_clause

body ::= proper_body | body_stub

proper_body ::=   subprogram_body | package_body |
                  task_body | protected_body

body_stub ::=  subprogram_body_stub | package_body_stub |
               task_body_stub | protected_body_stub

subprogram_body_stub ::=
        subprogram_specification 'is'  'separate' ";"
```


1.4.1.9 Blockkonzept, lokale und globale Namen

Blöcke sind Anweisungen, die mit einem (evtl. leeren) Deklarationsteil beginnen, dem eine (nicht-leere) Folge von Anweisungen folgt.

Die in einem Block definierten Namen leben ab der Deklarationsstelle bis zum Verlassen des Blockes.

Jedem Block wird ein neuer Speicherbereich im (lokalen) Kellerspeicher des Programms zugeordnet, in dem die neu eingeführten Variablen und Konstanten abgelegt werden. Mit Verlassen des Blocks wird dieser Speicherbereich frei gegeben.

Namen können in Unterblöcken neu deklariert werden. Jede Verwendung eines Namens bezieht sich stets auf die Deklaration im kleinsten umfassenden Block. Durch Neudeklarationen werden diese Namen in den Unter-Blöcken ungültig oder unsichtbar (obwohl sie weiter leben).

Jeder Name, der in einem Block definiert wird, heißt lokal bzgl. dieses Blocks. Namen, die in Ober-Blöcken definiert und in Unter-Blöcken verwendet werden, heißen in den Unter-Blöcken global.

1.4.1.10 Ausnahmen (exceptions): In der Syntaxdefinition von Blöcken tritt in Ada 95 weiterhin die Ausnahmebehandlung (exception_handler) auf. Wir betrachten auch diese genauer.

Die zugehörigen Deklarationen der Namen "exception_name" erfolgt in der "exception_declaration" im Deklarationsteil des Blocks (siehe oben unter basic_declaration):

```
exception_declaration ::=
        defining_identifier_list ":"  'exception' ";"

defining_identifier_list ::=
        defining_identifier {"," defining_identifier}

defining_identifier ::= identifier
```



```
exception_handler ::=
        'when' [choice_parameter_specification ":"]
        exception_choice {| exception_choice} "=>"
        sequence_of_statements

choice_parameter_specification ::= defining_identifier

exception_choice ::= exception_name | 'others'
```

Die Ausnahmebehandlung am Ende eines Blocks hat also die Form

```
exception
   when <Name des Fehlers>  =>  <Anweisungsfolge>
   when <Name des Fehlers>  =>  <Anweisungsfolge>
   .....
   when <Name des Fehlers>  =>  <Anweisungsfolge>
   when others  =>  <Anweisungsfolge>
```

Wenn wir später Moduln oder andere Programmeinheiten definieren, so werden wir dort auch Fehlermöglichkeiten festlegen. Wenn zum Beispiel von einem leeren Keller gelesen werden soll (top(empty) ist undefiniert), so kann man dies abfangen und beider Definition von "top" eine Ausnahme

```
Lesefehler: exception
```

deklarieren und diese dann mittels (Sprachelement raise)

```
if isempty then  raise Lesefehler;
else  "arbeite mit top(Keller) weiter"  end if;
```

auslösen. Dort, wo fehlerhafte Kellerzugriffe erfolgen können, legt man dann eine Ausnahmebehandlung der Form

```
exception
when Lesefehler =>  Put("Zugriff auf leeren Keller");
                    < weitere Anweisungen >;
when others => Put("Unbekannter Fehler"); ...
```

fest, um die Fehler *kontrolliert* abzufangen.

In Ada sind vier Standardfehler voreingestellt:

Constraint_Error: Überschreitung eines Bereichs

Program_Error: Nicht ausführbare Anweisung

Storage_Error: Verfügbarer Speicherplatz überschritten

Tasking_Error: Fehler bei nebenläufiger Verarbeitung

In der Regel verwendet man diese voreingestellten Fehler und gibt am Ende eines Blocks an, was beim Auftreten dieser Fehler geschehen soll. Formuliert man dagegen keine Ausnahmebehandlung, so bricht das Programm beim Auftreten eines dieser Fehler mit einer Fehlermeldung ab.

In den vielen Ada-Programmen finden sich daher Blöcke mit Ausnahmeregeln der folgenden Form:

```
declare ... ;
begin ...
......
exception
   when Constraint_Error  =>  X := 24; Y:= ... ;
   when Storage_Error  =>  ...;
   when others  =>  ...;
end;
```


1.4.1.11: Reservierte Wörter

Im Prinzip könnte man beliebige Wörter bzw. Identifikatoren für Namen verwenden. (In Sprachen wie PL/I ist dies zugelassen.) In vielen Sprachen sind jedoch einige Namen "reserviert", d.h., sie haben eine feste Bedeutung und dürfen nicht umdefiniert werden.

In Ada 95 sind dies folgende 37 bereits vorgestellten Namen:
abs, access, and, array, begin, case, constant, declare, delta, digits, do, else, elsif, end, exception, for, if, is, loop, mod, new, not, null, of, or, others, raise, range, record, rem, reverse, subtype, then, type, when, while, xor.

32 reservierte Wörter kommen in Ada 95 noch hinzu:
abort, abstract, accept, aliased, all, at, body, delay, entry, exit, function, generic, goto, in, limited, out, package, pragma, private, procedure, protected, renames, requeue, return, select, separate, tagged, task, terminate, until, use, with.

Man darf also in Ada Wörter wie integer, storage_error, boolean, float usw. für eigene Definitionen verwenden. Die vordefinierte Bedeutung geht dann innerhalb des jeweiligen Blocks verloren.