Typische Fallen

1. Konstanten als aktuelle Parameter von Unterprogrammen, insbesondere wenn die Konstante durch
   `PARAMETER (NMAX=100)`
   „aussieht“ wie eine Variable.
   Wenn der Konstanten-Speicherplatz auf dem Rechner „geschützt“ ist, gibt es den Fehler
   `segmentation fault`
   beim versuchten Überschreiben der Konstanten im Unterprogramm. Anderenfalls erhält die „vermeintliche“ Konstante einen neuen Wert ...

2. Vergleiche zwischen Gleitkommazahlen (insbesondere berechneten) sollten nicht durch exakte Tests `.EQ.` oder `.NE.` erfolgen. Rundungsfehler im Bereich des Maschinenepsilons sollte man tolerieren:
   `ABS(A-B) .LT. eps` (`eps`$> \varepsilon_M$), oder besser
   `ABS(A-B) .LT. eps*MAX(ABS(A),ABS(B))`

3. Schleifen mit reeller Laufvariable `DO I=E1,E2,E3 .` (in Fortran77 zugelassen)

   Die Schleife wird `N`-mal ausgeführt, wobei `N = MAX( 0, INT( (E2-E1+E3)/E3) )`.
   Empfindliche Reaktion gegenüber Rundungsfehlern.

4. `MOD`-Funktion für reelle Argumente kann unerwünschte Ergebnisse liefern, z.B. berechnet `MOD(1.0,0.2)` evtl. `0.2` statt `0.0`.

   Grund: `MOD(R1,R2):=R1-(R2*INT(R1/R2))`. Die Zahl `0.2` kann im Rechner nicht exakt gespeichert werden (im Unterschied zu `1.0`). Je nach Rundungsverhalten ist daher `R1/R2` < 5.0 oder > 5.0.

5. Zuweisungen von reellen Ausdrücken an ganzzahlige Variable („auf solche Momente warten die Rundungsfehler nur“). Kritisch ist es, Indizes aus reellen Ausdrücken zu berechnen.

6. Standardfunktionen (intrinsic) sind i.a. generisch, aber richten sich nicht nach dem Variablentyp in der Zuweisung, z.B.:

   ```
   integer I
   double precision X
   I=123456789
   X=REAL(I)
   ```

   Hier konvertiert `REAL` eben nicht in `doubleprecision`, sondern zuerst in `real` und erst bei der Zuweisung an `X` wird daraus `doubleprecision`, was zu Genauigkeitsverlust führt (Ergebnis: `X=123456780.`) Ähnliche Probleme gibt es bei allen (z.T. geschachtelten) mathematischen Typkonvertierungen.

7. Index überschreitet vereinbarte Grenzen eines Feldes. Verheerende Folgen sind oft erst später an einer ganz anderen Stelle des Programms zu spüren.
   Wenn möglich, sollte mit Bereichsgrenzenüberprüfung compiliert werden, bis das Programm ausreichend getestet ist.

8. Inhalt von Common-Blöcken kann zur Laufzeit verlorengehen, wenn in der Unterprogrammhierachie keines der Unterprogramme aktiv ist, in denen der Common-Block vereinbart ist.

9. Der Fortran-Standard verbietet es, dieselbe Variable an zwei verschiedenen Stellen der Argumentliste an ein Unterprogramm zu übergeben:

   ```
   call SUB1(A,A)
   ...
   subroutine SUB1(X,Y)
   ...
   ```

Bspw. ist es durchaus bequem, ein vorhandenes Unterprogramm
`subroutine vdaxpy(n,z,x,alpha,y)`
für $z_i = x_i + \alpha y_i$ so aufzurufen:
`call vdaxpy(n,x,x,alpha,y)`,
um $x_i = x_i + \alpha y_i$ akkumulierend zu berechnen. Das funktioniert fast immer, ist aber formal nicht erlaubt.

Ebensowenig darf eine Common-Variable als Argument an ein Unterprogramm übergeben werden, das diesen Common-Block ebenfalls benutzt.
Falsche Ergebnisse können in solchen Fällen auch durch „zu gute“ Optimierung entstehen.

10. Es gibt keine Garantie für eine bestimmte Reihenfolge der Berechnung von (gleichberechtigten) Teilausdrücken eines arithmetischen Ausdrucks, außer dass geschachtelte Klammerausdrücke berücksichtigt werden.

    Man kann die tatsächliche Reihenfolge der Berechnung z. B. so testen:

    ```
    program test
    integer F,I
    I=F(1)+F(2)+F(3)+F(4)
    ...
    end
    integer function F(k)
    write(*,*) 'F(',k,')'
    F=k
    return
    end
    ```

11. Eine verkürzte Auswertung von logischen Ausdrücken wird nicht garantiert, insbesondere nicht, wenn Funktionsaufrufe darin enthalten sind (wegen möglicher Seiteneffekte in diesen Aufrufen). Bspw. kann folgender Test zum Abbruch wegen negativem Argument der `log`-Funktion führen, obwohl unabhängig vom zweiten Teil der ganze Ausdruck `.FALSE.` ist.

    ```
    IF( (X .GT. 0.0) .AND. (LOG(X) .LE. 1.23) ) THEN
    ```

12. Bei trigonometrischen Funktionen ist die korrekte Transformation von sehr großen Argumenten $|x| \gg \pi$ auf das Intervall $-\pi \ldots \pi$ fraglich.

13. Abweichende Datentypen in aktueller Parameterliste der `call`-Anweisung und formaler Parameterliste der `Subroutine`-Anweisung. Es steht nur eine Bitfolge des Wertes im Speicher, keine Typ-Information. Fehlinterpretation des Speicherinhalts führt zu extremen Zahlenwerten (oder `NaN`).

Gibt es trotzdem noch Vorteile gegenüber anderen Programmiersprachen?

- zahlreiche generische Standardfunktionen (meist assembler-codiert)
- komplexe Arithmetik, `**`-Operator
- in F90: Feldmanipulationen, Teilfelder, Erweiterungen für Parallelisierung
- Sprach-Design von Fortran erlaubt maximale Geschwindigkeit (starke Optimierung), warum?
  - keine expliziten Pointer (Pointer machen für den Compiler den späteren Speicherzugriff undurchsichtig); in F90 sind Pointer möglich, wenn man auf Optimierung teilweise verzichten kann;
  - Prinzip der (relativ) statischen Speicherallokierung erspart internen Organisationsaufwand zur Laufzeit;
  - ausschließliche Parameterübergabe „by reference“ ist meist effektiver
- (numerische) Software-Entwicklung der letzten 50 Jahre ist in zahlreichen Fortran-Quellen dokumentiert und verfügbar, siehe z.B. www.netlib.org/
- einfacher lesbar als bspw. C (?) - vor allem für „Non-Experts“
  aber:
  „A poor programmer can generate bad code in any language.“