Informatik I Übung, Woche 40

Giuseppe Accaputo

1. Oktober, 2015

Plan für heute

1. Nachbesprechung Übung 2
2. Vorbesprechung Übung 3
3. Zusammenfassung der für Übung 3 wichtigen Vorlesungsslides

Syntaxfehler

- **Syntaxfehler**: Fehler im Programmcode, die durch unkorrektes Verwenden der Sprache erzeugt und vom Compiler während der Kompilierung erkannt werden
  - Beispiele: fehlender Strich-Punkt zwischen zwei Anweisungen, ungültiger Variablennamen
- **Analogie**: Artikel für ein Magazin schreiben
  - Author des Artikels $\Leftrightarrow$ Programmierer
  - Artikel $\Leftrightarrow$ Programmcode
  - Person, welche beim Magazin für die Veröffentlichung des Textes verantwortlich ist und den Text vor der Veröffentlichung auf grammatikalische Fehler überprüft $\Leftrightarrow$ Compiler
  - Grammatikalische Fehler im Artikel $\Leftrightarrow$ Syntaxfehler

Welche Fehler erkennt der Compiler nicht?

- **Semantische Fehler**: Fehler in der Bedeutung des Programms
  - Beispiel: Wir führen eine `Anweisung` nur dann aus, wenn der Benutzer über 18 Jahre alt ist, schreiben jedoch im Programmcode

```
IF alter < 18 THEN Anweisung;
```

Der Compiler wird diesen Code kompilieren da er syntaktisch korrekt ist, jedoch ist er semantisch falsch (`Anweisung` wird für Benutzer ausgeführt, welche unter 18 Jahre alt sind).

Der in dieser Situation semantisch korrekte Code würde wie folgt aussehen:

```
IF alter > 18 THEN Anweisung;
```

IF-Abfrage

- Eine IF-Abfrage ist eine Anweisung; deshalb wird sie mit einem Strich-Punkt abgeschlossen sollte eine weitere Anweisung darauf folgen

```
IF ... THEN Anweisung;
```

```
IF ... THEN Anweisung_1
ELSE Anweisung_2;
```

```
IF ... THEN Anweisung_1
ELSE IF ... THEN Anweisung_2
ELSE Anweisung_3;
```

Aufgabe 2.1: Prog2.pas

Frage: Welches sind die Syntaxfehler (die Fehler, welche vom Compiler erkannt werden) im folgenden Codeausschnitt?

```
...
  IF y < 0 THEN
    x := 1; z := 3;

  IF y = 0 THEN x:=5;
  ELSE x := 6;
...
```

Aufgabe 2.1: Prog2.pas und Semantik

```
...
  IF y < 0 THEN
    BEGIN
        x := 1;
        z := 3;
    END

  IF y = 0 THEN x:=5
  ELSE x := 6;
...
```

IF-Abfrage Beispiel

Frage: Ist der folgende Code-Ausschnitt syntaktisch korrekt? Wenn ja, was hat die INTEGER Variable a für einen Wert nach der Ausführung des Code-Ausschnitts?

```
...
a := 4;
IF a = 5 THEN ;
    a := 8;
```

IF-Abfrage Beispiel

Frage: Ist der folgende Code-Ausschnitt syntaktisch korrekt? Wenn ja, was hat die `INTEGER` Variable a für einen Wert nach der Ausführung des Code-Ausschnitts?

```
...
a := 4;
IF a = 5 THEN ;
    a := 8;
...
```

Antwort: `a := 8`. Das Semikolon direkt nach `THEN` bedeutet, dass eine leere Anweisung ausgeführt werden soll wenn a=5 gilt. a:=8 wird daher immer ausgeführt, unabhängig davon ob a den Wert 5 hat.

Reminder: Präzedenz und Assoziativität

Präzedenz:

- Boole'sche Operatoren haben höhere Präzedenez als Vergleichsoperatoren:
  1. Klammern: `(...)`
  2. Unäre Operatoren: `+`, `-`, `not`
  3. Punkt-Operatoren: `*`, `/`, `div`, `mod`, `and`
  4. Strich-Operatoren: `+`, `-`, `or`, `xor`
  5. Vergleichs-Operatoren: `=`, `<>`, `<`, `>`, `<=`, `>=`

Assoziativität:

- Linksassoziativ: `A op B op C` $\Leftrightarrow$ `((A op B)op C)`
  - Punkt-, Strich- und Vergleichs-Operatoren
- Rechtsassoziativ: `A op B op C` $\Leftrightarrow$ `(A op (B op C))`
  - Unäre Operatoren

Präzedenz und Assoziativität Beispiel I

Frage: Ist der folgende Code-Ausschnitt syntaktisch korrekt? Wenn ja, was hat die `BOOLEAN` Variable `b` für einen Wert nach der Ausführung des Code-Ausschnitts?

```
...
b := not (22 > 0) and (8/2 < 5);
...
```

Präzedenz und Assoziativität Beispiel I

Frage: Ist der folgende Code-Ausschnitt syntaktisch korrekt? Wenn ja, was hat die `BOOLEAN` Variable `b` für einen Wert nach der Ausführung des Code-Ausschnitts?

```
...
b := not (22 > 0) and (8/2 < 5);
...
```

Antwort: `b := true`

Präzedenz und Assoziativität Beispiel II

Frage: Ist der folgende Code-Ausschnitt syntaktisch korrekt? Wenn ja, was hat die `BOOLEAN` Variable c für einen Wert nach der Ausführung des Code-Ausschnitts?

```
...
c  := 7  >= 5  >= 3;
...
```

Präzedenz und Assoziativität Beispiel II

Frage: Ist der folgende Code-Ausschnitt syntaktisch korrekt? Wenn ja, was hat die `BOOLEAN` Variable c für einen Wert nach der Ausführung des Code-Ausschnitts?

```
...
c := 7 >= 5 >= 3;
...
```

Antwort: Der Code-Ausschnitt ist syntaktisch falsch.

Grund: Der <= Operator ist links-assoziativ, d.h. `c := (7 >= 5)>= 3;`. Da `(7 >= 5)` zu `true` evaluiert, wird versucht ein `BOOLEAN` mit einem `INTEGER` zu vergleichen, jedoch verlangt der <= Operator zwei Variablen vom gleichen Typ.

Präzedenz und Assoziativität Beispiel III

Frage: Ist der folgende Code-Ausschnitt syntaktisch korrekt? Wenn ja, was hat die BOOLEAN Variable d für einen Wert nach der Ausführung des Code-Ausschnitts?

```
...
VAR
     e : REAL;
     f : INTEGER;
...
e := 10.023; f := 2;
d := e >= f;
...
```

Präzedenz und Assoziativität Beispiel III

Frage: Ist der folgende Code-Ausschnitt syntaktisch korrekt? Wenn ja, was hat die `BOOLEAN` Variable `c` für einen Wert nach der Ausführung des Code-Ausschnitts?

```
...
VAR
     e : REAL;
     f : INTEGER;
...
e := 10.023; f := 2;
d := e >= f;
...
```

Antwort: `c := true`, weil die `INTEGER` Variable `f` implizit in ein `REAL` umgewandelt wird vom Compiler.

Präzedenz und Assoziativität Beispiel IV

```
VAR
    a,b,c,d : INTEGER;
BEGIN
    a := 1; b := 2; c := 3; d := 4;
    IF a < b and c < d THEN
        Writeln('Yes')
    ELSE
        Writeln('No');
END.
```

Frage: Wie sieht die Ausgabe auf der Konsole aus?

Präzedenz und Assoziativität Beispiel IV

Antwort: Wegen der Präzedenz der Boole'schen Operatoren meldet der Compiler einen Fehler, da zuerst `b and c` evaluiert wird, jedoch `and` nur mit `BOOLEAN` Variablen arbeiten kann. Um den Fehler zu beheben werden Klammern um `a < b` und `c < d` platziert:

```
VAR
    a,b,c,d : INTEGER;
BEGIN
    a := 1; b := 2; c := 3; d := 4;
    IF (a < b) and (c < d) THEN
        Writeln('Yes')
    ELSE
        Writeln('No');
END.
```

De Morgansche Regeln

1. `not (a and b)` $\equiv$ `(not a) or (not b)`
2. `not (a or b)` $\equiv$ `(not a) and (not b)`

Short-Circuit Auswertung (Kurzschlussauswertung)

Regeln:

- `a and b`: Werte zuerst `a` aus; werte `b` nur aus falls `a` wahr ist
- `a or b`: Werte zuerst `a` aus; werte `b` nur aus falls `a` falsch ist

Beispiele:

- Folgende Werte sind gegeben:
  ```
  a := 1; b := 2; c := false;
  ```

  Was wird in den folgenden Zeilen ausgewertet?
  - `IF (a = b) and c THEN Anweisung_1;`
  - `IF (a = b) or c THEN Anweisung_2;`

WHILE-DO

```
WHILE Bedingung DO Anweisung;
```

- **In Worte:**
  SOLANGE Bedingung wahr ist, führe Anweisung aus
  - Programmierer ist verantwortlich dafür, dass die WHILE-Schleife nach einer endlichen Anzahl Iterationen abbricht

WHILE-DO ein: Beispiel

```
VAR
    a : BOOLEAN;
BEGIN
    a := true;
    WHILE a DO Writeln('Hello, world!')
END.
```

Frage: Wie sieht die Ausgabe auf der Konsole aus?

FOR-Schleife: Syntax

1.

```
FOR Variable := Startwert TO Endwert
DO Anweisung;
```

2.

```
FOR Variable := Startwert DOWNTO Endwert
DO Anweisung;
```

- **In Worte:**
  Zähle `Variable` von `Startwert` bis `Endwert` und führe jedes mal `Anweisung` aus

FOR-Schleife: Informationen zur Anwendung

- `Variable, Startwert` und `Endwert` müssen als `INTEGER` Variablen deklariert werden:

```
VAR
    Variable, Startwert, Endwert :
        ↪ INTEGER;
```

- `Anweisung` kann eine einzelne Anweisung sein oder ein Block bestehend aus mehreren Anweisungen

FOR-Schleife: Informationen zur Anwendung

- `Variable` ist eine *read-only* Variable, d.h. sie kann in der `FOR`-Schleife nicht verändert werden (nur lesen erlaubt).

Folgendes funktioniert also nicht:

```
FOR Variable := Startwert TO Endwert DO
    ↪ Variable := Variable + 1;
```

$\Longrightarrow$ `Variable` wird verändert: `Variable := Variable + 1`

Folgendes funktioniert stattdessen:

```
FOR Variable := Startwert TO Endwert DO
    ↪ NeueVariable := Variable + 1;
```

$\Longrightarrow$ `Variable` wird nur gelesen (read-only)

FOR-Schleife: ein Beispiel

```
VAR
    start_v, end_v, curr_v : INTEGER;
BEGIN
    start_v := 1;
    end_v := 5;

    FOR curr_v := start_v TO end_v
    DO Writeln(curr_v);
END.
```

- Wie sieht die Ausgabe auf der Konsole aus?

REPEAT-UNTIL

```
REPEAT Anweisung_1; Anweisung_2; ...;
    ↪ Anweisung_N; UNTIL Bedingung;
```

- **In Worte:**
  Wiederhole die Anweisungssequenz
  `Anweisung_1, Anweisung_2, ..., Anweisung_N` bis
  `Bedingung` erfüllt ist

REPEAT-UNTIL: Beispiel

Fordere den Benutzer auf, für das Geschlecht entweder `m` oder `w` einzugeben. Gibt der Benutzer stattdessen einen anderen Buchstaben ein, so wird er solange aufgefordert das Geschlecht einzugeben, bis er korrekterweise entweder `m` oder `w` eingegeben hat:

```
REPEAT
    Write('Bitte␣Geschlecht␣eingeben␣(m/w):␣
        ↪ ');
    Readln(input);
UNTIL (input = 'm') or (input = 'w');
```

Wann soll ich welche Schleife verwenden?

- Verwende eine `FOR`-Schleife wenn die Anzahl der Iterationen *bekannt* ist
- Verwende eine `WHILE-DO`-Schleife wenn die Anzahl der Iterationen *unbekannt* ist
- Unterschied zwischen `WHILE-DO` und `REPEAT-UNTIL`:
  - `WHILE-DO` überprüft zuerst ob `Bedingung` erfüllt ist und führt erst dann die `Anweisung` aus
    - `WHILE-DO`-Schleife wird abgebrochen, wenn `Bedingung` **nicht erfüllt** ist
  - `REPEAT-UNTIL` führt die `Anweisung` mindestens einmal aus und überprüft erst dann, ob die `Bedingung` erfüllt ist
    - `REPEAT-UNTIL`-Schleife wird abgebrochen, wenn `Bedingung` **erfüllt** ist

FOR-TO → WHILE-DO

Jede FOR-TO-Schleife kann in eine WHILE-DO-Schleife umgewandelt werden (Umkehrung gilt nicht immer)

```
FOR Variable := Startwert TO Endwert
DO Anweisung;
```

ist äquivalent zu

```
Variable := Startwert;

WHILE Variable <= Endwert DO
    BEGIN
        Anweisung;
        Variable := Variable + 1;
    END
```