Übungsblatt 0 zu “Programmiersprachen”

Berthold Hoffmann, Studiengang Informatik (hof@tzi.de)

Besprechung am 25.10.04

Was sind Zählschleifen?

Untersuche *Zählschleifen* in den Referenzsprachen *Ada*, *Eiffel* und *Java*. (Da Schleifen auf Zuständen basieren, macht es wenig Sinn, sie in den funktionalen Referenzsprachen *Haskell* und *SML* oder in *Prolog* zu suchen.)

Als einmaliger Service des Veranstalters (*Applaus, bitte!*) sind hier schon die passenden Auszüge aus den Sprachbeschreibungen von Ada und Eiffel eingefügt. Die Definition von Zählschleifen in *Java* sollte vertraut sein, oder es sollte klar sein, wo die Definition zu finden ist.

Zählschleifen in Ada

[Aus: *Ada 95 Reference Manual*, Abschnitt 5.5 *Loop Statements*, `http://www.adahome.com/rm95/`]

A loop_statement includes a sequence_of_statements that is to be executed repeatedly, zero or more times.

Syntax

```
loop_statement ::=
    [loop_statement_identifier:]
        [iteration_scheme] loop
            sequence_of_statements
        end loop [loop_identifier];

iteration_scheme ::= while condition
    | for loop_parameter_specification

loop_parameter_specification ::=
    defining_identifier in [reverse] discrete_subtype_definition
```

Anmerkung. Eine discrete_subtype_definition definiert eine Teilmenge eines diskreten Typs. In Ada sind die Aufzählungstypen und ganzzahlige Typen diskret (und deren Untertypen). Die Typen Character und Boolean sind Aufzählungstypen. Teilmengen können direkt angegeben werden durch einen Teilbereich (discrete_range) der Form **range** *ugr* .. *ogr*, wobei *ugr* und *ogr* Ausdrücke desselben diskreten Typs sind. Die Spezifikation **reverse** *range* besagt, dass die Elemente des Teilbereichs *range* in umgekehrter Reihenfolge durchlaufen werden.

If a loop_statement has a loop_statement_identifier, then the identifier shall be repeated after the **end loop**; otherwise, there shall not be an identifier after the **end loop**.

Static Semantics

A loop_parameter_specification declares a loop parameter, which is an object whose subtype is that defined by the discrete_subtype_definition.

Dynamic Semantics

For the execution of a loop_statement, the sequence_of_statements is executed repeatedly, zero or more times, until the loop_statement is complete. The loop_statement is complete when a transfer of control occurs that transfers control out of the loop, or, in the case of an iteration_scheme, as specified below.

For the execution of a loop_statement with a while iteration_scheme, the condition is evaluated before each execution of the sequence_of_statements; if the value of the condition is True, the sequence_of_statements is executed; if False, the execution of the loop_statement is complete.

For the execution of a loop_statement with a for iteration_scheme, the loop_parameter_specification is first elaborated. This elaboration creates the loop parameter and elaborates the discrete_subtype_definition. If the discrete_subtype_definition defines a subtype with a null range, the execution of the loop_statement is complete. Otherwise, the sequence_of_statements is executed once for each value of the discrete subtype defined by the discrete_subtype_definition (or until the loop is left as a consequence of a transfer of control). Prior to each such iteration, the corresponding value of the discrete subtype is assigned to the loop parameter. These values are assigned in increasing order unless the reserved word reverse is present, in which case the values are assigned in decreasing order.

NOTES

1. A loop parameter is a constant; it cannot be updated within the sequence_of_statements of the loop (see 3.3).

2. An object_declaration should not be given for a loop parameter, since the loop parameter is automatically declared by the loop_parameter_specification. The scope of a loop parameter extends from the loop_parameter_specification to the end of the loop_statement, and the visibility rules are such that a loop parameter is only visible within the sequence_of_statements of the loop.

3. The discrete_subtype_definition of a for loop is elaborated just once. Use of the reserved word **reverse** does not alter the discrete subtype defined, so that the following iteration_schemes are not equivalent; the first has a null range.

```
for J in reverse 1 .. 0
for J in 0 .. 1
```

Examples

. . .

Example of a loop statement with a for iteration scheme:

```
for J in Buffer'Range loop -- works even with a null range
    if Buffer(J) /= Space then
        Put(Buffer(J));
    end if;
end loop;
```

. . .

Zählschleife in Eiffel

[Aus: *Chapter H: An Eiffel Tutorial*, *Eiffel—The Language* von Bertrand Meyer, Seite 1009–1010, `http://www2.inf.ethz.ch/ meyer/ongoing/etl/`.]
. . .
The loop construct has the form

```
from
    initialization
until
    exit
invariant
    inv
variant
    var
loop
    body
end
```

[1] The invariant *inv* and variant *var* parts are optional, the others required. *initialization* and *body* are sequences of zero or more instructions; *exit* and *inv* are boolean expressions (more precisely, *inv* is an assertion); *var* is an integer expression.

The effect is to execute *initialization*, then, zero or more times until *exit* is satisfied, to execute *body.* (If after *initialization* the value of *exit* is already true, *body* will not be executed at all.) Note that the syntax of loops always includes an initialization, as most loops require some preparation. If not, just leave *initialization* empty, while including the **from** since it is a required component.

The assertion *inv*, if present, expresses a loop invariant (not to be confused with class invariants). For the loop to be correct, *initialization* must ensure *inv*, and then every iteration of body executed when *exit* is false must preserve the invariant; so the effect of the loop is to yield a state in which both *inv* and *exit* are true. The loop must terminate after a finite number of iterations, of course; this can be guaranteed by using a loop variant *var*. It must be an integer expression whose value is non-negative after execution of *initialization*, and decreased by at least one, while remaining non-negative, by any execution of *body* when *exit* is false; since a non-negative integer cannot be decreased forever, this ensures termination. The full-assertion-monitoring mode will check these properties of the invariant and variant after *initialization* and after each loop iteration, triggering an exception if the invariant does not hold or the variant is negative or does not decrease.
. . .
Beispiel (von Seite 1002):

```
local
    s: SAVINGS_ACCOUNT
do  from account_list start until account_list after loop
            s ?= acc_list item
                -- item from LIST yields the element at cursor position
        if s /= Void and then s interest_rate > Result then
            Result := s interest_rate
        end
        account_list forth
    end
end
```

Aufgabe

Schreibe folgende Arten von Zählschleifen in den Sprachen *Java*, *Ada* und *Eiffel* (wenn möglich):

1. Die Zählvariable durchläuft die Werte 1, 2, 3, 4, 5.
2. Die Zählvariable durchläuft die Werte 10, 20, 30, 40, 50.
3. Die Zählvariable durchläuft die Werte 50, 40, 30, 20, 10.
4. Eine nicht terminierende Zählschleife.

Beantworte die folgenden Fragen für beliebige Zählschleifen in den Sprachen *Java*, *Ada* und *Eiffel*:

1. Wie oft wird der Ausdruck ausgewertet, mit dem der Endwert für die Zählvariable festgelegt wird?
2. Kann die Zählvariable im Schleifenrumpf verändert werden?
3. Wovon hängt die Anzahl der Schleifendurchläufe ab? Ist sie vor der ersten Ausführung des Schleifenrumpfes bekannt?
4. Was passiert mit der Zählvariablen, nachdem die Schleife verlassen wurde? Welchen Wert hat sie?