Die Algorithmenbeschreibungssprache Jana

Günther Blaschek, Institut für Systemsoftware, JKU Linz
Juni 2011

Jana (Java-based Abstract Notation for Algorithms) ist eine Beschreibungssprache zur Formulierung von Algorithmen. Sie ist an die Programmiersprache Java angelehnt, aber einfacher und weitgehend frei von syntaktischem Ballast. Jana kommt mit wenigen Konstruktionen aus und lässt dem Benutzer Formulierungsfreiheiten.

Algorithmen

Jeder Algorithmus (Prozedur) hat einen Namen, Parameter und Anweisungen:

```
search (↓List l ↓int len ↓int x ↑int i) {
    Anweisungen
}
```

Bei den Parametern wird jeweils der Typ des Parameters vor dessen Namen angegeben. Spielt der Typ für den Algorithmus keine Rolle oder ist es eindeutig, welcher Typ gemeint ist, kann er auch weggelassen werden.

```
search (↓l ↓len ↓x ↑int i) { // die Typen von l, len und x sind unspezifiert
    Anweisungen
}
```

Pfeile werden verwendet, um zwischen Eingangs- (↓), Ausgangs-(↑) und Übergangsparametern (↕) zu unterscheiden. Die Pfeile können weggelassen werden, wenn nur Eingangsparameter vorkommen. Dann müssen die Parameter allerdings durch Beistriche getrennt werden. Bei Algorithmen mit Rückgabewerten (Funktionen) wird der Typ des Rückgabewerts vor den Namen des Algorithmus geschrieben:

```
int search (↓List l ↓int len ↓int x) {
    Anweisungen
    return i
}
```

Auch beim Aufruf eines Algorithmus müssen die Pfeile angegeben werden.

```
search(↓l ↓n ↓x ↑idx)
```

Kommentare

Kommentare beginnen mit zwei Schrägstrichen und erstrecken sich über den Rest der Zeile:

```
// dieser Kommentar erstreckt
// sich über zwei Zeilen
```

Operatoren

Zuweisung: =

Gleichheit: ==

Ansonsten wird bei den Operatoren große Freiheit gelassen. Es sollten, soweit vorhanden, möglichst die Java-Operatoren verwendet werden (!=, <, >, >=, <=, *, /, %, +, -, !, &&, ||, usw.)

Anweisungen

In Jana gibt es Zuweisungen, Prozeduraufrufe, Anweisungen zur Ablaufsteuerung und Textanweisungen. Anweisungen können zur besseren Lesbarkeit mit Strichpunkten abgeschlossen werden, müssen es aber nicht.

```
variable= ausdruck        // Zuweisung
print(↓line)              // Prozeduraufruf
line= readF(↓file)        // Funktionsaufruf
```

Zur Ablaufsteuerung stehen **if**-, **switch**, **while-, for**- **repeat**- und **return**-Anweisungen zur Verfügung. Geschweifte Klammern zur Begrenzung von Blöcken *müssen* angegeben werden.

```
if (ausdruck) {
   Anweisungen
}
```

```
if (ausdruck1) {
   Anweisungen
} else if (ausdruck2) {
   Anweisungen
} else {
   Anweisungen
}
```

```
switch (ausdruck) {
   case ausdruck1: {
      Anweisungen
   }
   case ausdruck2: {
      Anweisungen
   }
   ...
   default: {
      Anweisungen
   }
}
```

Auf ein **case** kann nicht nur eine Konstante, sondern ein beliebig komplexer Ausdruck folgen. Der **default**-Zweig kann auch wegfallen. Zu beachten ist ferner, dass es keine **break**-Anweisung gibt. Nach der Abarbeitung eines **case**-Zweiges wird die **switch**-Anweisung immer verlassen.

```
switch (zahl) {
   case 0:     { s= "0" }
   case <0:    { s= "negativ" }
   case >0:    { s= "positiv" }
}
```

Schleifen:

```
while (ausdruck) {
   Anweisungen
}
```

```
repeat {
   Anweisungen
} until (ausdruck)
```

```
for (...) {
   Anweisungen
}
```

Die Anwendung der **for**-Schleife ist sehr frei. Wichtig ist, dass mit Hilfe einer Laufvariablen über einen bestimmten Wertebereich iteriert wird. Die Laufvariable selbst kann im Kopf der **for**-Schleife deklariert werden und ist somit nur innerhalb der **for**-Schleife gültig. Mögliche Schreibweisen:

```
// iteriert von i=1 bis n
for (i=1 … n) {
    Anweisungen
}
```

```
// i nimmt nacheinander alle Werte der Menge α an
for (i ∈ α) {
    Anweisungen
}
```

```
// iteriert über 2er-Potenzen
for (int i=2, 4, 8, 16, …, n) {
    Anweisungen
}
```

```
// absteigende Schleife (i=n, n-1, n-2, …, 0)
for (i=n, n-1, …, 0) {
    Anweisungen
}
```

Die **return**-Anweisung kann von jeder beliebigen Stelle aus den Algorithmus beenden. Ist der Algorithmus eine Funktion, ist der Ausdruck hinter **return** das Resultat des Algorithmus.

```
return
```

```
return ausdruck
```

Textanweisungen sind freie Prosatexte, die eine Aktion oder einen Ausdruck beschreiben:

```
if (statistische Rohdaten sind vorhanden) {
    berechne Mittelwert und Varianzen
}
```

In Ausdrücken können arithmetische und boolesche Operatoren, sowie Konstanten, Variablen und Funktionsaufrufe verwendet werden.

Elementare Datentypen

Als elementare Typen stehen ganze Zahlen, Gleitkommazahlen, Zeichen und boolesche Werte zur Verfügung:

```
int:      ..., -1, 0, 1, 2, ...
float:    ..., 0.0, ...
char:     'a', 'b', 'c', ...
boolean:  true, false
```

Variablendeklarationen haben die Form „typ name“:

```
int i
boolean a,b
```

Bei Konstanten werden zusätzlich die Anfangswerte und das Schlüsselwort final angegeben:

```
final int i= 5
final boolean a= true, b= false
```

Statische Variablen (Gedächtnisvariablen) werden durch das Schlüsselwort **static** deklariert. Statischen Variablen muss in der Deklaration ein Anfangswert zugewiesen werden.

```
static boolean start = true
```

Arrays

Gruppen von gleichartigen Elementen werden in Arrays zusammengefasst. Dabei sind einige Freiheiten erlaubt:

- Es kann eine Untergrenze und eine Obergrenze pro Dimension angegeben werden.
- Wenn nur eine Größe *n* angegeben wird, erstreckt sich der Indexbereich von 0 bis *n*-1.
- Größen/Indexgrenzen können ganz weggelassen werden, wenn sie nicht benötigt werden.
- Größenangabe/Indexgrenzen können auf die Typbezeichnung oder den Namen folgen.
- Dimensionen werden innerhalb einer eckigen Klammer durch Kommas getrennt.

```
int[1:n] feld1          // eindimensionales int-Feld von 1 bis n
int feld1[1:n]          // äquivalent zu obiger Deklaration
char feld2[n]           // char-Feld (Zeichenkette) von 0 bis n-1
int feld3[]             // int-Feld, Größe unbestimmt, unterer Index 0
int feld4[-2:]          // int-Feld, Größe unbestimmt, unterer Index –2
int feld5[:k]           // int-Feld, Größe unbestimmt, oberer Index k
int [0:k, 0:z] matrix   // zweidimensionales int-Feld
```

Der Elementtyp eines Arrays muss nicht unbedingt spezifiziert werden. Man verwendet dann die Schreibweise „Variablenname []“.

```
liste[0:k]              // eindimensionales Feld von 0 bis k, Elementtyp beliebig
```

Auf einzelne Elemente wird durch Indizierung zugegriffen:

```
int getSum(↓int[1:n] feld ↓int n) {
    int sum= 0;
    for (int i= 1 .. n) {
        sum= sum+feld[i];
    }
    return sum;
}
```

Zeichenketten

Zeichenketten werden wie *char*-Felder behandelt. Über die Art der Speicherung (vorangestellte Längenangabe oder Abschlusszeichen) ist nichts spezifiziert. Die vordefinierte Funktion *strLen*() liefert die Länge einer Zeichenkette. Zum lexikalischen Vergleich von Zeichenketten können die Vergleichsoperatoren verwendet werden, genauso der Zuweisungsoperator zum Kopieren von Zeichenketten.

Benutzerdefinierte Datentypen

Neben den elementaren Typen sind auch benutzerdefinierte Datentypen erlaubt. Diese werden mit dem Schlüsselwort **type** angegeben. Typdefinitionen sind z.B.:

```
type Person = {                                    // Strukturtyp
    int id
    char[1:n] lastName, firstName
}

type Day = ( Monday, Tuesday, Wednesday,           // Aufzählungstyp
             Thursday, Friday, Saturday, Sunday )

type Matrix = int[7, 3]                            // Matrix mit Grenzen 0..6 und 0..2

type Month = (1..12)                               // Bereichstyp
```

Verwendung:

```
Person p
Day d
Matrix m
Month month

p.id = 5
p.lastName = "Maier"
p.firstName = "Christine"
d = Friday
m[2, 1] = 5
month = 11                                         // nicht erlaubt wäre: month = 13
```

Namenskonventionen

Es ist nützlich, bei der Namensgebung gewisse Konventionen einzuhalten, um die Lesbarkeit zu erhöhen. Variablen- und Prozedurnamen sollen mit Kleinbuchstaben, Namen von benutzerdefinierten Typen mit Großbuchstaben beginnen. Innerhalb der Namen kann Groß- und Kleinschreibung zur Strukturierung verwendet werden.

Weitere Schreibweisen

Werden zusätzliche Konstruktionen zur Beschreibung von Algorithmen gebraucht, so sollten diese an die Programmiersprache Java angelehnt und, wenn erforderlich, erläutert werden.

Standardfunktionen irgendeiner anderen Programmiersprache sollten nicht verwendet bzw. zumindest erläutert werden.

Folgende Funktionen können immer ohne Erläuterung verwendet werden:

```
read(↑value ↑boolean eof)   // Generische Funktion zum Lesen vom Eingabemedium;
                            // eof gibt an, ob das Ende d. Eingabestroms erreicht wurde;
                            // eof kann auch weggelassen werden
write(↓value)               // Schreiben auf das Ausgabemedium;
                            // der Typ von value ist beliebig
writeLn()                   // Zeilenvorschub auf dem Ausgabemedium
int ord(↓char ch)           // Liefert den Ordinalwert des Zeichens ch
                            // (z.B. ASCII- oder Unicode-Nummer)
char chr(↓int v)            // Liefert das Zeichen mit dem Ordinalwert v
                            // (z.B. ASCII- oder Unicode-Zeichen)
int strLen(↓char[] str)     // Liefert die Länge der Zeichenkette str
```

Beispiele

Es folgen zwei Beispiele für die Formulierung von Algorithmen mit Jana.

Lineares Suchen

```
suche(↓liste[1:n] ↓int länge ↓x ↑int i) {
    int j

    j= länge
    while (j>0 && liste[j]!=x) {
        j= j-1
    }
    i= j
    // 1≤i≤länge && x==liste[i] || i=0 && x ist nicht in liste enthalten
}
```

Entfernen mehrfacher Leerzeichen aus einer Zeichenkette

```
removeBlanks(↓char[1:n] source ↓int n ↑char[1:m] target ↑ int m) {
    m= 0
    for (int i= 1..n) {
        if (i==1 || source[i]!=' ' || source[i-1]!=' ') {
            m= m+1
            target[m]= source[i]
        }
    }
}
```