Informatik I/II - Zusamenfassung

Patrick Pletscher

19. Oktober 2003

1 Die Sprache Oberon

1.1 Syntax

Module
Module = MODULE ident ";" [IMPORTLIST] DeclSeq [BEGIN StatSeq] END ident ".".

Deklarationen
DeclSeq = CONST ConstDecl ";" | TYPE TypDecl ";" | VAR VarDecl ";"ProcDecl ";"

Konstanten
ConstDecl = ident ["*"] "=" ConstExpression.

Variablen
VarDecl = IdentList ":" type.
IdentList = ident ["*"]"," ident ["*"].

Prozeduren
ProcDecl = ProcHeading ";" ProcBody ident.
ProcHeading = PROCEDURE ident ["*"] [FormalPars]. ProcBody = DeclSeq [BEGIN StatSeq] END.

Anweisungen
StatSeq = stat ";" stat.
stat = [assignment | ProcCall | IfStat | CaseStat | WhileStat | RepeatStat | LoopStat | ForStat | WithStat | EXIT | RETURN [expression]].

Zuweisung
assignment = designator ":=" expression.

Alternative
IfStat = IF expression THEN StatSeq ELSIF expression THEN StatSeq [ELSE StatSeq] END.

Wiederholungen
WhileStat = WHILE expression DO StatSeq END.
RepeatStat = REPEAT StatSeq UNTIL expression.

Ausdrücke
expression = SimpleExpression [relation SimpleExpression].
relation = "=" | "#" | "<" | "≤" | ">" | "≥" | IN | IS.
SimpleExpression = ['+'|'-'] term AddOperator term.
AddOperator = "+" | "-" | OR.
term = factor MulOperator factor.
MulOperator = "*" | "/" | DIV | MOD | "&".
factor = number | CharConstant | string | NIL | set | designator [ActualParameters] | "(" expression ")" | " " factor.
set = "" [element "," element] "".
element = expression [".." expression].

1.2 Semantik

SHORTINT ⊂ INTEGER ⊂ LONGINT ⊂ REAL ⊂ LONGREAL

1.3 Arrays

Array = Serie von Elementen gleichen Typs
Deklaration
- Eindimensional: ARRAY M OF T
- Mehrdimensional: ARRAY M,N OF T

Elementzugriff a[i] bzw. b[i,j]
Offene Arrays als Parameter
- PROCEDURE P(a: ARRAY OF T)
- PROCEDURE Q(b: ARRAY OF ARRAY OF T)
- Aktuelle Längen LEN(a) bzw. LEN(b,i),i=1,2

Beispiel: Lexikografischer Vergleich

```
VAR pool: ARRAY 10000 OF CHAR;
PROCEDURE LE (i,j:INTEGER): BOOLEAN;
BEGIN
 WHILE (pool[i] # 0X) & (pool[j] # 0X) &
  (pool[i] = pool[j]) DO INC(i); INC(j);
 END;
 RETURN pool[i] <=  pool[j];
END LE;
```

1.4 Records

RecordType = RECORD ["("BaseType")"] FieldListSeq END.
FieldListSeq = FieldList ";" FieldList.
FieldList = [IdentList ":" type].
IdentList = identdef "," identdef.

Beipiel Person, Employee, Student

```
TYPE
 Person = RECORD
```

```
  name: ARRAY MaxNameLen OF CHAR;
  address: ARRAY MaxAddrLen OF CHAR;
  age: INTEGER;
 END;
 Student = RECORD (Person)
  semester, rank: INTEGER;
 END;
 Employee = RECORD (Person)
  function, salary: INTEGER;
 END;
```

Feldzugriffe p.name
p(Employee).salary ≥ 500

2 Rekursion

Permutationen

```
PROCEDURE Perm (p,n: INTEGER);
VAR i: INTEGER;
BEGIN
 IF p = n THEN
  FOR i := 0 TO n-1 DO WRITELN(Items[i]); END;
 ELSE
  FOR i := 0 TO n-1 DO
   Swap(Items[p], Items[i]);
   Perm(p+1, n);
   Swap(Items[p], Items[i]);
  END;
 END;
END Perm;

BEGIN
 Perm(0, 4);
 ...
```

3 Suchen und Sortieren

3.1 Suchen

Linear

```
i:=0;
// mit Sentinel
WHILE A[i] < t DO INC(i) END;
```

Binär

```
i:=-1;j:=N;
WHILE j # i+1 DO m:=(i+j) DIV 2;
 IF A[m]<=t THEN i:=m
 ELSE j:=m;
 END;
END;
```

3.2 Sortieren

Internes Sortieren durch sukzessives Einfügen

```
PROCEDURE Sort;
 VAR i,j: INTEGER; new: T;
BEGIN
 j:=1;
 WHILE j # N DO
  new := A[j]; i:=j;
  WHILE A[i-1] > new DO
   A[i] := A[i-1];DEC(i);
  END;
  A[i]:=new;
  INC(j);
 END;
END Sort;
```

Quicksort

```
PROCEDURE Quicksort(L,R: INTEGER);
 VAR i,j,x,t: INTEGER;
BEGIN
 i:=L;j:=R;x:=a[(L+R) DIV 2];
 LOOP
  WHILE a[i] < x DO INC(i) END;
  WHILE a[j] > x DO DEC(j) END;
  IF i >= j THEN EXIT END;
  t:=a[i]; a[i]:=a[j]; a[j]:=t;
  INC(i);DEC(j);
 END;
 IF i=j THEN INC(i); DEC(j); END;
 IF L < j THEN Quicksort(L,j) END;
 IF i < R THEN Quicksort(i,R) END;
END Quicksort;
```

3.3 Hashing statt Ordnen

Typiche Hashfunktion: **h(s)= s MOD prim**
Kollisionsbehandlung durch offene Adressierung:
h(s) = (s + ((prim-2) -s MOD (prim-2))) MOD prim

Erkennen von Schlüsselwörtern
Für jeden String $s = \ldots c_2 c_1 c_0$ sei
$h(s) = (c_0 + c_1 \cdot b + c_2 \cdot b^2 + \ldots)$ MOD p

4 Backtracking Algorithmen / Graphtraversierung

4.1 Rucksackproblem

```
// av = noch zu erreichender Wert
PROCEDURE Traverse(i:INTEGER; s:SET; av,w:REAL);
BEGIN
 // Gegenstand einschliessen
 IF av-val[i] > vmax THEN
  IF i = n-1 THEN opts:=s; vmax:=av-val[i]; END
  ELSE Traverse(i+1,s,av-val[i],w);
  END;
 END;
 // Gegenstand ausschliessen
 IF (w+weight[i] <= wmax) & (av-val[i] > vmax) THEN
  INCL(s,i);
  IF i=n-1 THEN opts:=s; vmax:=av; END
  ELSE Traverse(i+1,s,av,w+weight[i]);
  END;
```

```
 END;
END Traverse;

BEGIN
 vmax=0;opts={};
 Traverse(0,{},v[0]+v[1]+...+v[n-1],0)
 ...
```

4.2 Depth-First Strategie am Beispiel Travelling Salesman

```
PROCEDURE Visit(no,i: INTEGER; c: REAL);
 VAR j: INTEGER;
BEGIN
 visited[i] := TRUE;
 IF no = N-1 THEN
  IF c+a[i,0] < min THEN min:= c+a[i,0] END
 ELSE
  FOR j:=0 TO N-1 DO
   IF ~visited[j] & (c+a[i,j] < min) THEN
    Visit(no+1,j,c+a[i,j]) END;
  END;
 END;
 visited[i] := FALSE;
END Visit;

BEGIN
 FOR i:=0 TO N-1 DO
  visited[i] := FALSE;
 END;
 min := MAX(REAL);
 Visit(0,0,0);
 ...
```

Optimierungen

Konstruktion eines Minimalgerüstes nach Kruskal
K(min. Tour) > K(Gerüst durch Weglassen einer Kante) ≥ K(Minimalgerüst)

Registrierung von Traversierungen
Ersetze visited durch visitedFrom

Kantentabelle
Adjazenzmatrizen typischerweise "dünn besiedelt", darum besser eine Kantentabelle.

4.3 Breadth-First Strategie am Beispiel Chinesische Landesvermessung

Datenstruktur mit Agenda als Heap

```
TYPE Event = RECORD
 at: REAL;
 in, from: INTEGER;
END;

a: ARRAY NofConnections OF Event;
conn: ARRAY NofVillages+1 OF INTEGER;
dest: ARRAY NofConnections OF INTEGER;
time: ARRAY NofConnections OF REAL;
arrAt: ARRAY NofVillages OF REAL;
arrFrom: ARRAY Nof Villages OF INTEGER;

PROCEDURE Enter (at:REAL; in,from:INTEGER);
 VAR i: INTEGER;
BEGIN
 INC(N); i:=N;
 WHILE (i#1) & (at < a[i DIV 2].at) DO
  a[i]:=a[i DIV 2]; i:=i DIV 2;
 END;
 a[i].at:=at; a[i].in:=in; a[i].from:=from;
END Enter;

PROCEDURE GetNext(VAR next: Event);
 VAR i,j: INTEGER; last: Event;
BEGIN
 next:=a[1]; last:=a[N]; DEC(N); i:=1;
 LOOP
  IF i>N DIV 2 THEN EXIT END;
  j:=2*i;
  IF (j<N) & (a[j+1].at < a[j].at) THEN INC(j) END;
  IF a[j].at >= last.at THEN EXIT END;
  a[i]:=a[j];i:=j;
 END;
 a[i]:=last;
END GetNext;

PROCEDURE Swarm(i:INTEGER);
 VAR j: INTEGER;
BEGIN
 FOR j:= conn[i] TO conn[i+1]-1 DO
  Enter(arrAt[i]+time[j], dest[j], i);
 END;
END Swarm;

VAR
 e: Event;

BEGIN
 FOR i:=0 TO NofVillages-1 DO
  arrAt[i]:=MAX(REAL)
 END;
 // sentinel
 Enter(MAX(REAL), NofVillages, NofVillages);
 arrAt[0]:=0; Swarm(0); GetNext(e);
 WHILE e.at < MAX(REAL) DO
  IF arrAt[e.in] = MAX(REAL) THEN
   arrAt[e.in] := e.at; arrFrom[e.in] := e.from;
  END;
 END;
END.
```

5 Listen

5.1 Zeigerbasierte Objekte

Deklaration

```
    TYPE
```

```
MyObject= OBJECT
 x: XType;
 ..
END;
VAR obj: MyObject;
```

Erzeugung ("Instanzierung")
NEW(obj);

Benutzung
obj.x

5.2 Listen

Definition

```
TYPE
 TList = OBJECT
  next: TList;
  x: XType;
 END;
```

Einfügen
Am Kopf:

```
new.next:=first;first:=new;
```

Am Ende:

```
IF last # NIL THEN last.next:=new
ELSE first:=new END;
last := new;
```

Geordnet (mit End-Sentinel):

```
IF first.x <= new.x THEN cur := first;
 WHILE cur.next.x <= new.x DO
  cur:=cur.next
 END;
 new.next := cur.next; cur.next := new
ELSE new.next:=first; first:=new;END;
```

First-In-First-Out

```
PROCEDURE Arrive(new: Object);
BEGIN
 IF last # NIL THEN last.next:=new
 ELSE first:=new END;
 last := new;
END Arrive;
PROCEDURE Serve(VAR obj: Object);
BEGIN
 IF last = NIL THEN obj := NIL
 ELSE
  obj := first;
  IF first = last THEN last:=NIL
  ELSE first:=first.next END;
 END;
END Serve;
```

Last-In-First-Out

```
PROCEDURE In(i:INTEGER; new: Object);
BEGIN new.next:=first[i]; first[i]:= new;
END In;

PROCEDURE Out(i:INTEGER; VAR obj:Object);
BEGIN obj:=first[i];
 IF first[i] # NIL THEN
  first[i] := first[i].next END
END Out;
```

6 Simulation diskreter Systeme

Zustandsänderung nur durch explizite Ereignisse zu *diskreten Zeitpunkten*

6.1 Client-Server-Systeme

```
TYPE
 Object = OBJECT
  next: Object;
  t: REAL;
 END;

 Client = OBJECT (Object)
  arrT: REAL (*arrival time*)
 END;

 Server = OBJECT (Object)
  curC: Client (*current client*)
 END;

VAR
 firstC, lastC: Object; (*client waiting line*)
 firstS: Object; (*server pool*)
 cal: Object; (*event calendar*)

 nofC: INTEGER; totTime, simEnd, now: REAL;

 x: Object; c: Client; s: Server;

 PROCEDURE PutC(c: Client);
 PROCEDURE GetC(VAR c: Client);
 PROCEDURE PutS(s: Server);
 PROCEDURE GetS(VAR s: Server);
 PROCEDURE Enter(x: Object; t: REAL);
 PROCEDURE GetNext(VAR x: Object);

PROCEDURE Arrive(x: Client);
 VAR c: Client; s: Server;
BEGIN
 x.arrT := now;
 GetS(s);(* get server from pool*)
 IF s#NIL THEN
  s.curC:=x.Enter(s,now+exponential)
 ELSE PutC(x) (*put x into client waiting line*)
 END
 // schedule next arrival
 NEW(c);Enter(c,now+expontial);
END Arrive;
```

```
PROCEDURE End(x: Server);
 VAR s: Server; c: Client;
BEGIN
 INC(nofC);
 totTime := totTime+now-x.curC.arrT;
 GetC(c); (*get next client*)
 IF c # NIL THEN
  x.curC := c; Enter(x, now+exponantial)
 ELSE PutS(x); (*return server*)
END End;

lastC:=NIL;(*client waiting line*)
firstS := NIL; NEW(s); PutS(s);
NEW(s); PutS(s);
NEW(cal); cal.t:=MAX(REAL);
nofC:=0; totTime:=0; simEnd:=1000;
NEW(c); Enter(c,0); GetNext(x);
WHILE x.t <= simEnd DO
 now := x.t;
 IF x IS Client THEN Arrive(x(Client))
 ELSE End(x(Server)) END;
 GetNext(x);
END;
WRITELN(totTime/nofC);
```

7 Topologisches Sortieren

Internalisierung

```
read pred;
WHILE not end-of-input DO
 locate or create and insert pred;
 read succ;
 locate or create and insert succ;
 create and attach link to pred;
 connect link with succ;
 increment ref count in succ;
 read pred;
END
```

Externalisierung

```
WHILE member list not empty DO
 remove elem min with minimal ref count;
 write min;
 IF ref count of min # 0 THEN
  write "cycle"
 END;
 cur := min.succ;
 WHILE cur # NIL DO
  decrement ref count of cur.to
 END;
END
```

8 Bäume

8.1 Unix File Directories

```
TYPE
 Node = OBJECT
  next, down: Node;
  type, loc: INTEGER;
  name : ARRAY 30 OF CHAR;
 END;
```

8.2 Binärbäume

```
TYPE
 Node = OBJECT
  left, right: Node;
  d: AnyType;
 END;
```

Knotenweise Baumoperationen

```
PROCEDURE Optree (t: Tree);(*Preorder*)
BEGIN
 IF t # NIL THEN
  Opnode(t); Optree(t.left); Optree(t.right);
 END;
END Optree;

PROCEDURE Optree (t: Tree);(*Inorder*)
BEGIN
 IF t # NIL THEN
  Optree(t.left); Opnode(t); Optree(t.right);
 END;
END Optree;

PROCEDURE Optree (t: Tree);(*Postorder*)
BEGIN
 IF t # NIL THEN
  Optree(t.left); Optree(t.right); Opnode(t);
 END;
END Optree;
```

Expression-Trees: Auswertung

```
TYPE
 Node = OBJECT
  left, right: Node;
  op: INTEGER; val: REAL
 END;

PROCEDURE Val(x: Node): REAL;
BEGIN
 IF x.right = NIL THEN
  IF x.left = NIL THEN RETURN x.val
  ELSE RETURN Op1(x.op, Val(x.left));
  END;
 ELSE RETURN Op2(x.op, Val(x.left), Val(x.right));
 END;
END Val;
```

8.3 Geordnete Bäume (Search Trees)

Definition

Binärbaum B geordnet:
Für alle Knoten x in B gilt
$L(x) \leq x.key \,\&\, R(x) > x.key$

```
TYPE
 Node = OBJECT
  left, right: Node;
  key: INTEGER;
 END;

PROCEDURE Search(VAR r: Node; key: INTEGER);
BEGIN
 IF r # NIL THEN
  IF key < r.key THEN
   r := r.left;
   Search(r, key)
  ELSIF key > r.key THEN
   r := r.right;
   Search(r, key);
  END;
 END;
END Search;

BEGIN
 r := root; Search(r, key);
 WHILE r # NIL DO
  ...
  r := r.left; Search(r, key);
 END;
 ..
END;

PROCEDURE Insert(VAR r: Node; new: Node);
BEGIN
 Ir r = NIL THEN r := new
 ELSIF new.key <= r.key THEN
  Insert(r.left, new)
 ELSIF new.key > r.key THEN
  Insert(r.right, new)
 END;
END Insert;

PROCEDURE Remove(VAR r: Node; key: INTEGER);
VAR s,t: Node;
BEGIN
 IF r = NIL THEN (*key not in tree*)
 ELSIF key < r.key THEN Remove(r.left, key)
 ELSIF key > r.key THEN Remove(r.right, key)
 ELSE
  IF r.left = NIL THEN r:=r.right
  ELSIF r.right = NIL THEN r:=r.left
  ELSE
   s := r.left;
   IF s.right = NIL THEN
    s.right := r.right; r:= s
   ELSE
    t := s.right;
    WHILE t.right # NIL DO
     s:=t; t:=t.right
    END;
    s.right := t.left; t.left := r.left;
    t.right := r.right; r := t;
   END;
  END;
 END;
END Remove;
```

8.4 2/3/4 Bäume

Bei jedem Knoten 2,3 oder sogar 4 Unterbäume und 1, 2 oder 3 Root-Nodes, so dass gilt:
2-er: $L < x < R$
3-er: $L < x < M < y < R$
4-er: $L < x < ML < y < MR < z < R$

Ausgleichendes Einfügen in einen 2/3/4-Rot/Schwarz-Baum

```
TYPE
 Node = OBJECT
  left, right: Node;
  red: BOOLEAN;
  key: REAL;
 END;

PROCEDURE Insert(key: REAL; r: Node);
VAR
 f,g,h: Node; (*path-memory*)
BEGIN
 REPEAT
  h:=g; g:=f; f:=r;
  IF key < f.key THEN r:= f.left
  ELSE r:=f.right END;
  IF r.left.red & r.right.red THEN (*4-node*)
   r:= Split(h,g,f,r);
  END;
 UNTIL r = sentinel;
 NEW(r); r.key:=key; r.left:=s; r.right:=s;
 IF key < f.key THEN f.left:=r ELSE f.right:=r END;
 r:=Split(h,g,f,r);
END Insert;

PROCEDURE Split(h,g,f,r: Node):Node;
BEGIN
 r.left.red := FALSE;
 r.right.red := FALSE; r.red := TRUE;
 root.right.red := FALSE;
 IF f.red THEN
  IF (f=g.left) # (r=f.left) THEN
   IF f=g.left THEN f:=Rotate(f,r); g.left := f
   ELSE f:=Rotate(f,r); g.right:=f;
   END;
  END;
  IF g = h.left THEN g:=Rotate(g,f); h.left:=g
  ELSE g:= Rotate(g,f); h.right:=g;
  END;
  r:=g;
 END;
 RETURN r;
END Split;

PROCEDURE Rotate(VAR a,b: Node):Node;
BEGIN
 IF b=a.left THEN (*rotate right*)
```

```
  a.left:=b.right; b.right:=a
 ELSE (*rotate left*) a.right:=b.left; b.left:=a
 END;
 b.red := a.red := TRUE;
 RETURN b;
END Rotate;
```