Ausdrücke

Infix-Notation	Prefix-Notation	Postfix-Notation
$a * (b + c)$	$* a + b c$	$a b c + *$

Operator-Assoziativität und -Preceedence:

$a + b * c$	entwpricht	$a + (b * c)$	(Pascal, C, ...)
$a = b < c$	entwpricht	$(a = b) < c$	(Pascal)
$a == b < c$	entwpricht	$a == (b < c)$	(C)

Einige Sprachen unterstützen benutzerdefinierte Operatoren.

Bedingte Ausdrücke:

(a > b) ? a : b	(C)
if a > b then a else b	(ML)
case x of 1 => f1(y) | 2 => f2(y) | _ => g(y)	(ML)

Bedingte Anweisungen

if x > 0 **then** **if** x < 10 **then** x := 0 **else** x := 1000
if x > 0 **then** **begin** **if** x < 10 **then** x := 0 **end** **else** x := 1000
if x > 0 **then** **if** x < 10 **then** x := 0 **end** **else** x := 1000 **end**
if a **then** S1 **else** **if** b **then** S2 **else** **if** c **then** S3 **else** S4 **end**

```
switch (operator) {
   case '+': result = operand1 + operand2; break;
   case '-': result = operand1 - operand2; break;
   default:  break; }
```

case OPERATOR **is**
   when '+' => result := operand1 + operand2;
   when '−' => result := operand1 − operand2;
   when others => **null**;
end case

Schleifen

for var := lower **to** upper **do** statement	(Pascal)
for (int i = 0; i < 10; i++) {...}	(C++)
for var **in** discrete_range **loop** body **end loop**	(Ada)
while condition **do** statement	(Pascal)
while (expression) statement;	(C)
while condition **loop** loop_body **end loop**	(Ada)
repeat statement **until** condition	(Pascal)
do statement **while** (expression);	(C)
loop statement; **exit when** condition **end loop**	(Ada)

A: **loop** ... **loop** ... **exit** A; ... **end loop** ... **end loop** A;

Routinen

procedure p(**var** x: T; y: Q; **function** f(z: R): integer);

```
void proc(int* x, int y) {
   *x = *x + y; }
```

```
void proc(int& x, int y) {
   x = x + y; }
```

Alias-Probleme: Lesbarkeit der Programme wird erschwert
Optimierungen werden verhindert

Beispiel: u := x + z + f(x,y) + f(x,y) + x + z

Euclid vermeidet Alias-Probleme gänzlich
→ semantische Einschränkungen, dynamische Prüfungen

Ausnahmebehandlung in Ada

Vordefinierte Ausnahmen: Constraint_Error
Program_Error
Storage_Error
Tasking_Error

Benutzerdefinierte Ausnahmen (Bsp): Help: **exception**;

Expliziter Aufruf einer Ausnahme (Bsp): **raise** Help;

begin −− Block mit Ausnahmebehandlung
. . . statements . . .
exception when Help => . . . statements . . .
when Constraint_Error => . . . statements . . .
when others => . . . statements . . .
end;

Implementierung in SIMPLESEM

Ausnahmen erhalten vom Compiler eindeutige Namen.

Wenn eine Ausnahme aufgetreten ist, wird der entsprechende Handler dynamisch (entlang des “dynamic link”) gesucht.

Jeder “activation record” enthält einen Zeiger auf eine statische Tabelle; jeder Tabelleneintrag assoziiert einen Ausnahmenamen mit dem Rumpf des entsprechenden Handlers.

Es ist möglich, daß eine Ausnahme über dessen Scope hinaus propagiert wird.

Ausnahmebehandlung in C++

Beliebige Daten werden propagiert.

Aufruf einer Ausnahme:
throw Help(MSG1);

Deklaration propagierter Funktionen:
void foo() **throw**(Help, Zerodivide);

Spezielle Funktionen: unexpected()
terminate()

Ausnahmebehandlung in Java

Alle propagierten Ausnahmen müssen deklariert werden:
void foo() **throws** Help;

Form eines “try”-Blocks:
try
block;
catch(exception_type_1)
handler_1;
. . .
catch(exception_type_n)
handler_n;
finally
final_block;

Ausnahmebehandlung in ML

```
exception Neg

fun fact(n) =
  if n < 0 then raise Neg
  else if n = 0 then 1
    else n * fact(n - 1)

func fact_0(n) =
  fact(n) handle Neg => 0;
```


Pattern Matching

```
datatype day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
fun day_off(Sun) = true
  | day_off(Sat) = true
  | day_off(_) = false

fun reverse(nil) = nil
  | reverse(head::tail) = reverse(tail) @ [head]

fun rev(nil) = nil
  | rev(0::tail) = [0] @ [tail]
  | rev(head::tail) = rev(tail) @ [head]
```


Nichtdeterminismus und Backtracking

```
A if B or
     C or
     D;
C if E and
     F and
     G;
D if I or
     H;
```


Ereignisgesteuerte Berechnungen

Einige Sprachen unterstützen "event-driven computations" direkt (Visual Basic, Visual C++, Tcl/Tk)

In aktiven Datenbanken werden Trigger verwendet:

```
on event
when condition
  do action

on insert in EMPLOYEE
when TRUE
  do emp_number++
```

Prozesse

task type SERVER **is**
  entry NEXT_REQUEST(NR: **in** REQUEST);
  entry SHUT_DOWN;
end SERVER;

type SERVER_PTR **is access** SERVER;

MY_SERVER: SERVER;

task CHECKER **is**
  entry CHECK(T: **in** TEXT);
  entry CLOSE;
end CHECKER;

HIS_SERVER_PTR: SERVER_PTR := **new** SERVER;

Semaphore

Semaphore sind low-level Synchronisationsmechanismen.

Auf jedem Semaphor s sind zwei atomare Aktionen definiert:

P(s): if $s > 0$ then $s = s - 1$
else suspend current process

V(s): if there is a process suspended on the semaphore
then wake up process
else $s = s + 1$

Vor dem Eintritt in einen kritischen Abschnitt, der durch s gesichert ist, wird P(s) aufgerufen, nach Beendigung V(s).

Operationen des Betriebssystemkerns

Betriebssystemkern ist ADT mit diesen Operationen:

enqueue: Queue $\times$ Descriptor $\rightarrow$ Queue
dequeue: Queue $\rightarrow$ Queue $\times$ Descriptor
empty: Queue $\rightarrow$ Boolean

Jeder Uhren-Interrupt führt diese Operation aus:

```
Suspend_and_Select()
  RUNNING = process_status;
  READY_QUEUE.enqueue(RUNNING);
  RUNNING = READY_QUEUE.dequeue();
  process_status = RUNNING;
```


Implementierung der Semaphore

```
Suspend_on_Condition(c)
  RUNNING = process_status;
  CONDITION_QUEUE(c).enqueue(RUNNING);
  RUNNING = READY_QUEUE.dequeue();
  process_status = RUNNING;

Awaken(c)
  RUNNING = process_status;
  READY_QUEUE.enqueue(RUNNING);
  READY_QUEUE.enqueue(
                  CONDITION_QUEUE(c).dequeue());
  RUNNING = READY_QUEUE.dequeue();
  process_status = RUNNING;
```

Implementierung der Monitore

“mutual exclusion” durch Abschalten von Interrupts während der Ausführung von Monitor-Prozeduren

```
Continue(c)
  RUNNING = process_status (interrupts enabled,
     program counter set to return point from monitor call);
  READY_QUEUE.enqueue(RUNNING);
  if not CONDITION_QUEUE(c).empty()
    READY_QUEUE.enqueue(
                 CONDITION_QUEUE(c).dequeue());
  RUNNING = READY_QUEUE.dequeue();
  process_status = RUNNING;
```


Implementierung von Rendezvous (1)

Jedes “entry” hat einen Descriptor mit diesen Feldern:

- O: Boolean; true if entry is open
- W: waiting queue (task descriptors of callers)
- T: descriptor of task owning entry
- I: pointer to first instruction of accept body

```
Call_Entry(e)
  RUNNING = process_status;
  DESCR(e).W.enqueue(RUNNING);
  if DESCR(e).O
    for all entries oe of DESCR(e).T do oe.O = false;
    RUNNING = DESCR(e).T;
    RUNNING.ip = DESCR(e).I;
  else RUNNING = READY_QUEUE.dequeue();
  process_status = RUNNING;
```


Implementierung von Rendezvous (2)

```
At_End_Of_Accept_Body(e)
  RUNNING = process_status;
  READY_QUEUE.enqueue(DESCR(e).W.dequeue());
  READY_QUEUE.enqueue(RUNNING);
  RUNNING = READY_QUEUE.dequeue();
  process_status = RUNNING;

Execute_Accept_Statement(e)
  if DESCR(e).W.empty()
    DESCR(e).O = true;
    DESCR(e).T = process_status;
    RUNNING = READY_QUEUE.dequeue();
    process_status = RUNNING;
  -- else simply continue executing the accept body
```


Implementierung von Rendezvous (3)

```
Execute_Select_Statement()
  LOE = list of open entries in selection;
  if LOE is empty raise Program_Error;
  else
    if DESCR(e).W.empty() (for all e in LOE)
      for all e in LOE do
        DESCR(e).O = true;
        DESCR(e).T = process_status;
      RUNNING = READY_QUEUE.dequeue();
      process_status = RUNNING;
    else
      choose an e with not DESCR(e).W.empty();
      proceed execution from instruction DESCR(e).I;
```