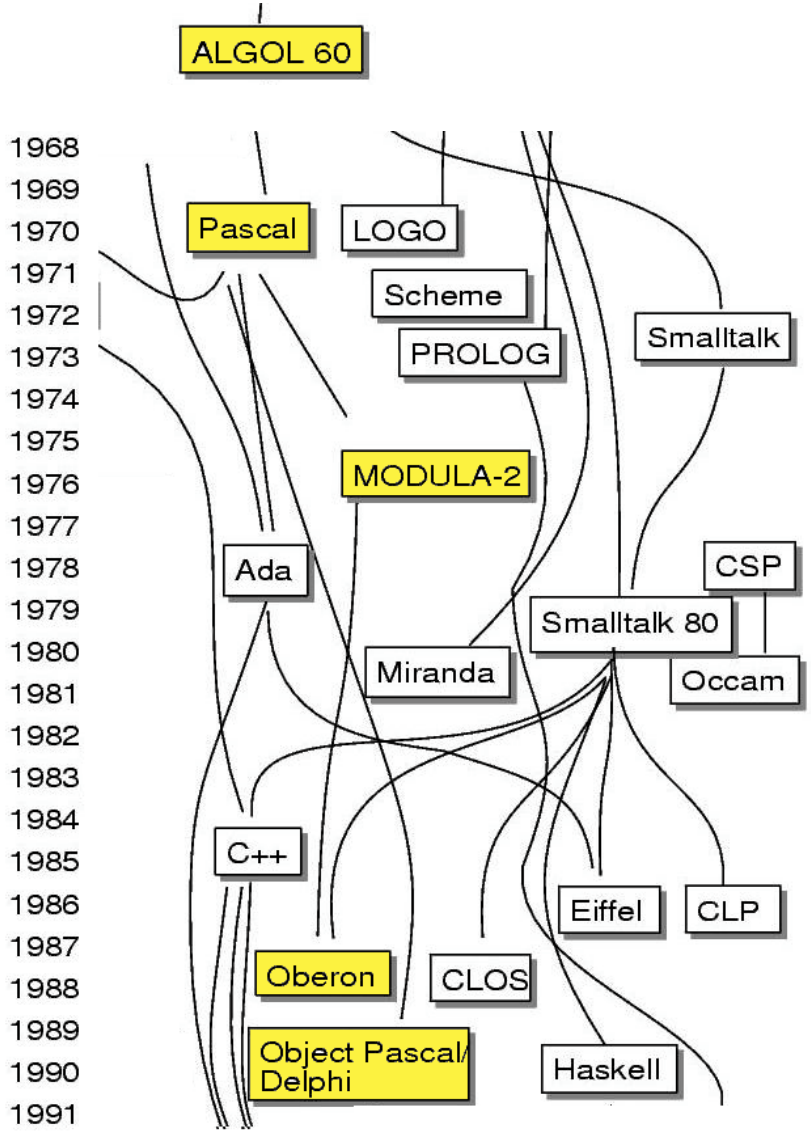
Pascal Familie - Modulkonzept

- Pascal Familie
- Lilith und Modula-2
- Die Sprache Modula-2

Pascal Familie im Staumbaum



Niklaus Wirth

- 1934: geboren in Winterthur
- 1954-1959: Studium der Elektrotechnik an der Eidgenössischen Technischen Hochschule (ETH) Zürich
- 1960-1963: Promotion an der University of California, Berkeley
- 1963-1967: Assistenz Professur an der UNIZ und der Stanford University
- 1968: Professor für Computerwissenschaften an der ETH Zürich
- 1976-1977: Visiting Researcher bei Xerox PARC
- 1977-1980: Entwicklung von Modula-2 und des Lilith Computers
- 1984: Auszeichnung mit dem Turing-Award
- 1984-1985: Visiting Researcher bei Xerox PARC
- 1985-1987: Entwicklung von Oberon und des Ceres Computers
- ab 1999: Emeritierung, weitere Ehrungen

Algol

Algol

- Wurde ab 1958 durch ein Komitee (mehrheitlich Mathematikern und N. Wirth) als saubere Programmiersprache definiert.
- Zwei Ziele
  - 1. Eine Sprache um Algorithmen zu beschreiben
  - 2. Die auf einem Computer ausgeführt werden können
- Konzepte waren - für die damalige Zeit - gut/sauber
- Wurde aber nie gross verwendet
  - nicht durch IBM unterstützt (damals die absolut dominierende Firma - so wie heute Apple, Google und Microsoft zusammen)
  - Compiler waren schwierig zu schreiben
    - *bei der Definition der Programmiersprachen Grammatik wurde nicht auf einfache Parsbarkeit geachtet*
  - Übersetzung dauerte z.T. Stunden

Pascal

Pascal 1970

- Überzeugung, dass Algol als Komiteesprache zu unhandlich
  - N.W. "ein Komitee ist in der Lage einen Entscheid zu fällen, der dümmer ist als jeder seiner einzelnen Mitglieder"
- Weiterentwicklung von Algol mit folgenden Zielen:
  - hoher Grad an Systematik
  - Verwendung mathematischer Notationen
  - einfache und intuitiv verständliche Regeln
  - hohe Effizienz
  - Sowohl für Wissenschaft als auch für Industrie geeignet
- N.W.: "...convinced that an elegant style and an effective implementation were not mutually exclusive..."
- Gut aber nicht perfekt: Mängel von Pascal
  - schwaches File I/O Konzept
  - Einfache nicht standardisierte Laufzeitbibliothek
- Durchsetzen konnten sich v.a. die späteren Versionen von Borland (Turbo Pascal, Delphi = objektorientiertes Pascal)

Pascal Hello World

- Hello World in Pascal

```
program HelloWorld;

begin
  writeln('Hello World');
end.
```

Pascal

- Selbst definierte Typen durch Records (aus Algol übernommen, im Vergleich zu Fortran aber eine Neuerung)

```
type Date = record
    day:   1..31;
    month: 1..12;
    year:  1900..2100;
end;

var today: Date;
```

- Durchgehend statische Typenprüfung

Pascal

- Pointer zu Records, Heap für dynamische Speicherallokation
- Aber keine Pointerarrithmetik wie in C

```
type Listpointer = ^Listelement;
     Listelement = record
                      Info: Content;
                           Next: Listpointer;
                   end;
...
var Element: Listpointer;
...
 New (Element); {Storage allocation on the heap}
```

Pascal

School of Engineering

- Erste Form von Polymorphismus durch Variant Record

```
type Coordinate = record
      case kind: (cartesian, polar) of
      cartesian: (x, y: real);
      polar: (r: real; phi: integer);
end;
```

Pascal, Quicksort Example

```
program Sort_Test;
const Max = 5000;
type Index = 1 .. Max;
type Table = array [Index] of Real;
var A: Table;

procedure Quick_Sort(First, Last:Index);

   procedure Swap (
        var I1, I2: Real);
        var H: Real;
   begin
       H := I1; I1:= I2;I2:= H;
   end; {Swap}

var I,J: Index;
    X  : Real;
```

local procedure

```
begin
   I:= First ; J:= Last;
   X:= A[(First+Last) div 2];
   repeat
      while A[I]<X do I:= I+1;
      while A[J]>X do J:= J-1;
      if I <= J then  begin
        Swap (A[I], A[J]);
        I:= I + 1;  J:= J-1;
      end;
   until I > J;
   if First < J then
      Quick_Sort (First, J);
   if Last  > I then
      Quick_Sort (I, Last);
end; {Quick_Sort}

begin
   Quick_Sort (1, Max);
end.
```

Modula 2

Modula-2 1979

- Sprache für Workstation Lilith

Wichtigste Neuerungen:

- Modularisierung und getrennte Kompilierung von Programmen
- Möglichkeit, bereits vorhandene Funktionen aus Bibliotheken einzubinden
- Deshalb auch für grosse Projekte geeignet
- Sprache soll auch ganze Systeme beschreiben können,
  - Daher u.a. Module zur Low-Level-Programmierung und Grafikprogrammierung
  - Nebenläufigkeit: Coroutines (threads)

Lilith Workstation

- ETH Eigenentwicklung Workstation ab 1977

- Inspiriert durch Xerox Alto Computer
  - Wurde von Xerox aber nicht verkauft
  - N.W. entwickelte daraufhin die Lilith
- Hochauflösender Bildschirm & Maus
  - erste Maus von Logitech

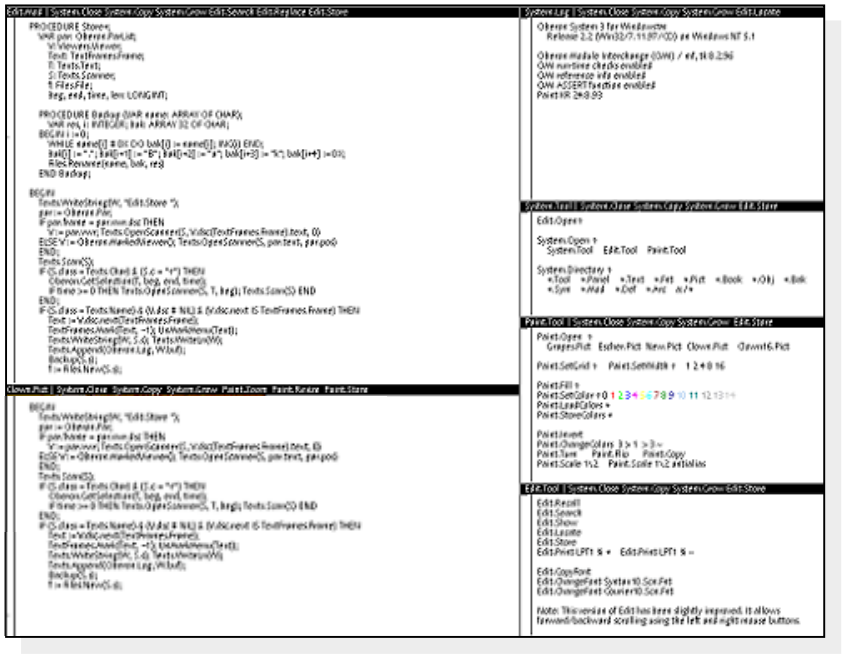
- Software war vollständig in Modula-2 geschrieben -> übersetzt in sog. M-Code
- Der M-Code wurde direkt in Hardware (Stack Maschine) ausgeführt
  - 16 Bit Bitslice Prozessor mit 7 MHz getaktet
  - Eigens entwickelter Microcode
  - Sehr kompakter Maschinencode (doppelt so kompakt wie vergleichbare Maschinen)
  - 3-7 Zyklen pro M-Code Instruktion

Neue Konzepte in Modula-2

- Information Hiding (Parnas): Der direkte Zugriff auf die interne Datenstruktur wird unterbunden und erfolgt stattdessen über definierte Schnittstellen (Black-Box-Modell).
  - Funktionen und Daten gekapselt in Module
  - Module werden getrennt kompiliert
  - Schnittstellenüberprüfung schon beim Kompilieren.
  - Konzept des abstrakten Datentyps
- Sicherheit durch Typisierung: Die Programmierfehler vermeiden, die entstehen können, wenn der Compiler gefährliche, implizite (nicht ausdrücklich ausgezeichnete) Typ-Konvertierungen automatisch durchführt.
  - Eine strenge und vollständige Typenprüfung.
  - Qualifizierte Zugriffe/Imports möglich
  - Pointer (sicher); keine Arithmetik, keine Zugriffe ausserhalb definiertem Bereich
- Für Betriebssystem geeignet
  - Nebenläufigkeit durch Coroutinen
  - Low-Level Konstrukte wie ADDRESS und BITSET

Oberon Sprache und System 1987

- ETH Grundausbildung bis ca. Jahr 2000, nachher Eiffel
- Neuerungen gegenüber Modula:
  - Typeninklusion und Kompatibilität
    - *in z.B. arithmetischen Ausdrücken*
  - Erweiterbarer (polymorpher) Recordtyp
  - Typentest
- Aber ***keine* typengebundene Methoden**



Oberon System

- Eigenwilliges Fenstersystem
- Erweiterbarer Recordtyp für Fenster und Meldungen verwendet
- Gekachelt in zwei Bereiche
  - *grosser "Inhaltsbereiche"*
  - *kleinerer "Toolbereich"*
- 2013 Version mit wichtigsten Quellen (Compiler, System)
  - *https://www.inf.ethz.ch/personal/wirth/ProjectOberon/index.html*
  - *https://schierlm.github.io/OberonEmulator/*

Einführung

Hello World

- Input/Output im Modul InOut

```
MODULE hello;
FROM InOut IMPORT WriteString, WriteLn;

BEGIN
    WriteString("Hello, world!");
    WriteLn;
END hello.
```

Reservierte Wörter, Kommentar, Konventionen

- Reservierte Wörter

AND	ELSE	LOOP	RETURN
ARRAY	ELSIF	MOD	SET
BEGIN	END	MODULE	THEN
BY	EXIT	NOT	TO
CASE	FOR	OF	TYPE
CONST	FROM	OR	UNTIL
DEFINITION	IF	POINTER	VAR
DIV	IMPLEMENTATION	PROCEDURE	WHILE
DO	IMPORT	RECORD	WITH
	IN	REPEAT	

- Kommentar in (* und *)
- Case Sensitive; Identifier meist gross geschrieben;
- CamelCase für Identifier :`Primzahl, Max`

Datentypen, Variablen und Konstanten

Vorzeichenbehafteter Ganzzahlentyp: INTEGER

- Wertebereich abhängig von Maschinenarchitektur
  - i.d.R. 16 Bit -32768 .. 32767
  - Grösse durch MIN(INTEGER) und MAX(INTEGER) zur Laufzeit bestimmbar
- Operationen
  - + : Addition
  - - : Subraktion
  - * : Multiplikation
  - / : Real Division
  - DIV : Integer Division
  - MOD : Divisionsrest
  - () : Klammerausdrücke
  - ABS : Absolut Betrag
  - ODD: Test ob Zahl ungerade
  - MAX (T) : Obere Grenze des Wertebereichs
  - INC(A), INC(A,N) : erhöhen um eins/ einen Wert
  - DEC(A), DEC(A,N): erniedrigen um eins/einen Wert

Vorzeichenloser Ganzzahlentyp: CARDINAL

- Wertebereich abhängig von Maschinenarchitektur
  - 16 Bit -> 0..65535
- Alle positiven ganzen Zahlen (einschliesslich 0).
  - Grösse durch MIN(CARDINAL) und MAX(CARDINAL) zur Laufzeit bestimmbar
- Operationen
  - gleiche wie für INTEGER
- Bemerkung:
  - Versuch eine negative Zahl in einer CARDINAL Variablen zu speichern führt zu einem Fehler
  - INTEGER und CARDINAL Werte können einander zugewiesen werden (mit :=)
  - **gemischte Ausdrücke sind aber NICHT erlaubt; explizite Umwandlung nötig**
- Beispiele

```
Int1 := INTGEGER(Card1) + Int1; (* mixed types     *)
Card2 := Card1 - 13 * 2 + CARDINAL(Int1);   (* mixed types     *)
```

Variablendeklaration & Zuweisung

- Eingeleitet durch Schlüsselwort **VAR**
- Variablennamen durch "," abgetrennt
- Typenbezeichnung durch ":" abgetrennt

- Beispiel

```
VAR Count : INTEGER;  (* The sum of two variables *)
    x,y   : INTEGER;  (* The two variables to add *)
```

Zuweisung :=

- Zuweisung mit ":="
- Beispiel

```
x := 1;
```

Konstanten

- Festlegung eines unveränderbaren Wertes
- Typ wird implizit festgelegt

- Beispiel

```
CONST PI = 3.1415;
```

Fliesskommazahlen Typ: REAL

- Wertebereich Maschinenabhängig

- Operationen
  - Im wesentlich dieselben wie für Ganzzahlen (ausser DIV und MOD)
  - FLOAT(i): Umwandlung eine Ganzzahl in REAL Wert
  - **TRUNC(r)**: Grösste Ganzzahl kleiner als Argument -> abrunden
  - **ENTIER(r)**: Kleinste Ganzzahl grösser als Argument -> aufrunden
  - ROUND(r): Runden auf nächste Ganzzahl

- Beispiel

```
A := FLOAT(Inumber);    (* INTEGER to REAL                *)
B := FLOAT(Cnumber);    (* CARDINAL to REAL               *)
Inumber := ENTIER(Sum);  (* REAL to INTEGER                *)
Cnumber := ENTIER(Sum);  (* REAL to CARDINAL               *)
```

Buchstaben Typ: CHAR

- Buchstaben in ASCII Kodieren
- werden in " " oder ' ' geschrieben
  - können z.B. verwendet werden um ein einzelnes " zu definieren `'"'`
- Konstanten als Dezimalwert gefolgt von "C"
- Beispiele

```
VAR ch, ch2 : CHAR;
ch := "a";
ch := 12C;
```

- Laufzeitumwandlungen

```
ch := CHR(i);
i := ORD(ch); (* liefert CARDINAL *)
ch2 := CAP(ch); (* Umwandlung in Grossbuchstabe *)
```

Keine impliziten Typenumwandlungen

```
Int1  := 14;
Int2  := 35;
Card1 := Int1 + Int2 + 23;              (* assignment compatible *)
Card2 := Card1 - 13 * 2 + CARDINAL(Int1);   (* mixed types     *)
Card2 := Card1 - 13 * 2 + CARDINAL(Int1);   (* assignment comp *)
Int1  := Int2 * INTEGER(Card1);
Real1 := 12.0;
Real2 := Real1 + FLOAT(Card2) * 1.112;     (* CARDINAL to REAL *)
Real2 := Real2 + FLOAT(CARDINAL(Int1));    (* INTEGER to REAL  *)
Int1  := TRUNC(Real1) + Int2 * 3;          (* Incompatible error 1 *)
Int1  := TRUNC(Real1) + CARDINAL(Int2) * 3;     (* error fixed *)
Int1  := INTEGER(TRUNC(Real1)) + Int2 * 3;      (* error fixed *)
Card1 := TRUNC(Real1) + Int2 * 3;          (* Incompatible error 1 *)
Card1 := INTEGER(TRUNC(Real1)) + Int2 * 3;      (* error fixed *)
Card1 := TRUNC(Real1) + CARDINAL(Int2) *3;      (* error fixed *)
Char1 := "A";
Int1  := ORD(Char1) + Int2;                (* Incompatible error 1 *)
Int1  := INTEGER(ORD(Char1)) + Int2;            (* error fixed *)
Int1  := ORD(Char1) + CARDINAL(Int2);           (* error fixed *)
Card1 := ORD(Char1) + Card1;
Real2 := FLOAT(ORD(Char1)) + 1.2345;
Char1 := CHR(TRUNC(FLOAT(ORD(Char1))));      (* Sheer Nonsense *)
```

Aufzählungstypen und Sets

Aufzählungstyp, Typendeklaration

- Aufzählung der Werte
- Die einzelnen Werte werden in "(", ")" durch "," getrennt aufgezählt
- Alle Werte automatisch im Namensraum definiert (ohne Präfix).

- Definition eines neuen Typs durch: **TYPE**

```
TYPE
   Material = (Eisen, Stahl, Kupfer, Glass, Gummi)
   Weekday = (Monday, Thursday, Wednesday, Thuersday, Friday,
   Saturday, Sunday);

VAR
   M : Material;
   Day : Weekday;

Day := Monday;
```

Unterbereichstypen

- Unterbereichstypen durch Wertebereichsgrenzen definiert
- Bei Berechnungen mit diesem Typ führen Bereichsüberschreitungen zu Laufzeit- oder Kompilationsfehlern.

- Beispiele

```
[0 .. 9];
["0" .. "9"];
["A" .. "Z"];
[0C .. 177C];
[0 .. 40000];
[-10 .. +10];
TYPE Weekday = [Monday .. Friday];
```

Ungültige Bereiche:

```
[ "A" ..  "9"]
[1.5 ..2.5]
```

Boolean

- Bool'sche Typ: BOOLEAN
- Ist der Aufzählungstyp mit den Werten (TRUE, FALSE)
- Operationen
  - OR : logische Konjuktion
  - AND : logische Disjunktion
  - NOT : Negation
- Beispiele

```
VAR IsIt, WillIt, What : BOOLEAN;
IsIt := TRUE;
What := FALSE;
WillIt := IsIt AND What;      (* FALSE because What is FALSE *)
WillIt := IsIt AND NOT What;  (* TRUE                        *)
WillIt := IsIt OR What;       (* TRUE because one is TRUE    *)
WillIt := NOT IsIt OR What;   (* FALSE                       *)
IsIt := (A = B) OR (B = C) OR (A = 22);
IsIt := ((A < B) AND (B < C)) OR NOT (B > C);
```

Vergleichsausdrücke Beispiele

- Boolsche Variablen in Kombination mit Vergleichen

```
VAR IsIt, WillIt, What : BOOLEAN;
   A, B, C             : INTEGER;

   A := 22;         (* Assign some values to work with *)
   B := 12;
   C := -12;

   IsIt := A = 22;     (* TRUE    - equal to                    *)
   IsIt := A = 23;     (* FALSE   - equal to                    *)
   WillIt := A > B;    (* TRUE    - greater than                *)
   WillIt := A < B;    (* FALSE   - less than                   *)
   IsIt := B <= 12;    (* TRUE    - less than or equal          *)
   IsIt := B >= 4;     (* TRUE    - greater than or equal       *)
   IsIt := A # B;      (* TRUE   -  not equal                   *)
```

SET

Von Aufzählungstypen und Bereichstypen können SETs gebildet werden

Beispiele

```
SET OF [3..5];
SET OF [Monday .. Sunday];
SET OF Weekday;
```

- Operationen
  - + : Oder Verknüpfung
  - - : Exklusion der Set Menge
  - * : Und Verknüpfung, Schnittmenge
  - / : Symmetrische Differenz
  - IN : Test ob einzelner Wert in Menge
  - INLC : Einschliessen eines Wertes in Set
  - EXCL: Ausschliessen eines Wertes aus Set

BITSET

Menge der Bits eines Maschinenworts

- BITSET = SET OF [0 .. WordSize - 1]

- Operationen gleich wie Set
- Spezielle Konstanten: `{1,4}, {1,4..6,15}`

Strukturierte Datentypen

Arrays

- Array Index Wertebereich werden einfach durch Unterbereichstyp angegeben

```
TYPE Name = ARRAY[1 .. 20] OF CHAR
TYPE Name2 = ARRAY[1 .. 19] OF CHAR
TYPE SALOD = ARRAY WEEKDAY OF REAL
```

- mehrdimensionale Arrays

```
TYPE Name2D = ARRAY[1 .. 20],[0 .. 19] OF CHAR
```

- Arrays mit unterschiedlichen Indexbereichen sind **nicht** zuweisungskompatibel

```
VAR N : Name;
N2: Name2;
N := N2; (* Fehler *)
```

Offene Arrays

- In Prozeduren (später) kann ein Array als Argument mit beliebiger oberer Grenze übergeben werden
- Mittels HIGH kann diese in der Prozedur überprüft werden

- Beispiel

```
PROCEDURE GenAddNumbers(Donkey : ARRAY OF CARDINAL);
VAR CountUp, Sum : CARDINAL;
BEGIN
   Sum := 0;
   FOR CountUp := 0 TO HIGH(Donkey) DO
      Sum := Sum + Donkey[CountUp];
   END;
   WriteCard(Sum,5);
   WriteLn;
END GenAddNumbers;
```

Recordtyp

- Aus mehreren einfachen Typen kann ein Recordtyp gebildet werden

- Beispiel

```
TYPE Datum = RECORD
     Tag :[1..31];
     Monat:[Jan .. Dez];
     Jahr :[1900..2006];
END
```

- Deklaration von Variablen von diesem Typ

```
VAR Date : Datum;
```

- Qualifizierter Zugriff mittels "."

```
Date.Tag := 1;
```

Pointer Typ

- POINTER ist an einen bestimmten Datentyp (seinen Basistyp) gebunden
  - NEW fordert Speicher an
  - DISPOSE gibt Speicher wieder frei
  - ^ derefenziert Pointer Variable
  - NIL als undefinierter Pointer
  - Wird auf Methoden von Storage umgeleitet
    - *ALLOCATE fordert Speicher an*
    - *TSIZE wird Grösse bestimmt; wichtig hier der richtige Typ*
    - *DEALLOCATE gibt Speicher wieder frei*
- Beispiel

```
TYPE Name = ARRAY[0..20] OF CHAR;
VAR  MyName : POINTER TO Name;    (* MyName points to a string *)
     MyAge  : POINTER TO INTEGER; (* MyAge points to an INTEGER
   NEW(MyAge);
   NEW(MyName);
   MyAge^ := 27;
   MyName^ := "John Q. Doe";
   DISPOSE (MyAge);
   DISPOSE (MyAge);
```

Kontrollstrukturen

Bedingte Anweisung: IF THEN

- IF, THEN, ELSE und ELSIF für bedingte Anweisung
- Abgeschlossen durch END

- Beispiel

```
IF Betrag < 0 THEN
   AenderungKontostand := TRUE;
  Soll := Soll + (-Betrag);
ELSIF Betrag > 0 THEN
  AenderungKontostand := TRUE;
  Haben := Haben + Betrag;
ELSE
  AenderungKontostand := FALSE;
END
```

Auswahl Anweisung: CASE

- Mit CASE wird die Auswahl Anweisung definiert

- Beispiel

```
CASE Operator OF
     '+': Ergebnis := Zahl1 + Zahl2 |
     '-': Ergebnis := Zahl1 - Zahl2 |
     '*': Ergebnis := Zahl1 * Zahl2 |
     '/': Ergebnis := Zahl1 / Zahl2 |
     ELSE Fehler := TRUE
END;
```

WHILE Schleife

- Schleife mit Abbruchbedingung
- Beispiel

```
WHILE a<b DO
      a:=a+1;
      b:=b-1;
END;
```

FOR Schleife

- Schleife mit Schleifenzähler

- Beispiel

```
FOR TestTeiler := 2 TO Zahl-1 BY 1 DO
     WriteCard(TestTeiler, 1);
     WriteLn;
END;
```

Loop

- Schleife ohne Abbruchbedingung
- EXIT um Schleife zu verlasen

- Beispiel

```
LOOP
    WriteLn;
    ReadInt(Zahl);
    Read(Antwort);
    Read(Zeichen);
    IF NOT (Antwort = JaAntwort) AND (Zeichen = EOL) THEN EXIT; END;
END;
```

Repeat Until

- Schleife mit Test am Schluss
- Beispiel

```
REPEAT
    WriteString("Bitte eine ganze Zahl und Return eingeben: ");
    Read(Antwort);
    Read(Zeichen);
UNTIL NOT (Antwort = JaAntwort) AND (Zeichen = EOL);
WriteString("Ende!");
```

Proceduren

Prozedur

- Eine Gruppe von Anweisungen kann zu einer Prozedur zusammengefasst werden
- Innerhalb der Prozedur können lokale Variablen deklariert werden

- Beispiel

```
PROCEDURE WriteMessage;
VAR X : INTEGER;
BEGIN
   WriteString("This is the message");
   WriteLn;
END WriteMessage;
```

- Aufruf

```
WriteMessage;
```

Prozedur Parameter

- Parameter können als Werte (Default) oder als Referenz (VAR) Parameter übergeben werden

- Beispiel

```
PROCEDURE AddTheFruit (Value1,Value2 : CARDINAL;  (* by value*)
                       VAR Total     : CARDINAL;  (* by reference *)
BEGIN
   Total := Value1 + Value2;
END AddTheFruit;
```

…

- Aufruf

```
AddTheFruit(Apple,Pear,Fruit);
WriteString("The total number of fruits is ");
WriteCard(Fruit,5);
```

Prozedur mit Rückgabewert, Funktion

- Eine Prozedur kann mit RETURN einen Wert zurückgeben

- Beispiel

```
PROCEDURE QuadOfSum(Number1, Number2 : INTEGER) : INTEGER;
BEGIN
   RETURN 4*(Number1 + Number2);
END QuadOfSum;
…
Dogs := 4;
Cats := 3;
Feet := QuadOfSum(Dogs,Cats);
```

Prozedur Typ und Variable

- Ein Prozedur Typ enthält (die Adresse) einer Prozedur
- Lediglich die Signatur des Typs wird festgelegt .

- Beispiel

```
TYPE Winkelfunktion = PROCEDURE (REAL):REAL;
VAR sin : Winkelfunktion;
```

Modulares Design

Module

- Module sind Einheiten die übersetzt, verteilt und gewartet werden
  - haben i.d. R. gröbere Granularität als Klassen
- Haben klar definierte Schnittstellen
- Sämtliche Interaktion zu andern Modulen findet **AUSSCHLIESSLICH** über diese Schnittstellen statt
- Schnittstellen sind somit der **"Vertrag"** zwischen
  - Anwender des Moduls
  - Implementator des Moduls
- Gute Modularisierung
  - maximale Kohäsion innerhalb des Module
  - minimale Abhängigkeit zwischen den Modulen -> schlanke Schnittstellen

Definition Modul

- Ein (externes) Definitionsmodul legt die Schnittstelle des Moduls fest
- Typen und Prozedur-Köpfe

- Diese sind in .DEF Module gespeichert

- Beispiel

```
DEFINITION MODULE ItemReader;

TYPE index = INTEGER;
     item = RECORD
               key: index
            END;
PROCEDURE ReadItem(): item;

END ItemReader.
```

Implementation Modul

- Im Implementationsmodul wird die Methode implementiert
- Es werden referenzierte Module importiert
  - Qualifiziert: IMPORT InOut -> Modulpräfix bei Gebrauch
  - Unqualifiziert: FROM InOut IMPORT WriteLine -> kein Modulpräfix nötig
- Im DEF Module deklarierte Typen müssen nicht nochmals deklariert werden
- Beispiel

```
IMPLEMENTATION MODULE ItemReader;
FROM InOut IMPORT WriteLn, WriteString, ReadInt, WriteInt;

PROCEDURE ReadItem(): item;
BEGIN …
END ReadItem;

BEGIN
(* Initialisierung *)
END ItemReader.
```

Main Modul

- Ein Module ohne Schnittstelle (= Hauptmodul)

- Beispiel

```
MODULE Test;
...

BEGIN
END Test.
```

Abstrakter Datentypen

- Definition eines Typnamens in Definitonsmodul ohne Bindung an einen konkreten Datentyp realisieren ADT.
- Deklaration eines abstrakten Datentyps
- auf POINTER-Typen beschränkt.

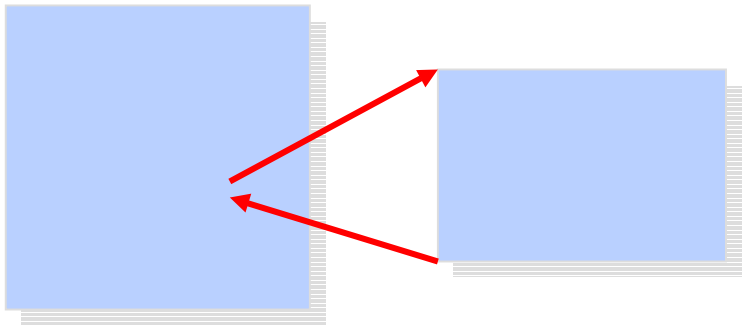
```
DEFINITION MODULE Test;
  TYPE Zahl;
PROCEDURE Readnum(Zahl);
END test.

IMPLEMENTATION MODULE Test;
  TYPE Zahl = POINTER TO Zahlfolge;
  Zahlfolge=RECORD
              vor:CARDINAL;
              nach:REAL;
  END;
```

Coroutinen

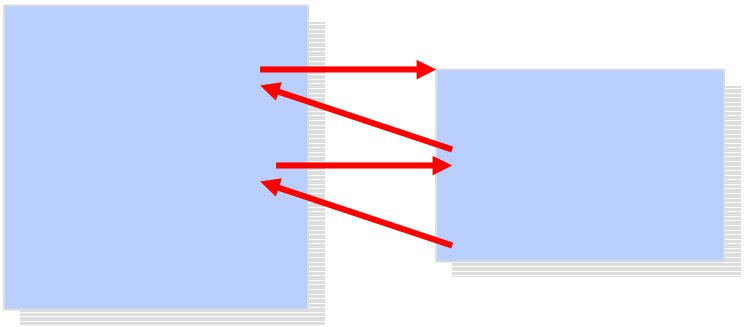
Coroutinen

- Verallgemeinerung des Konzepts einer Prozedur oder Funktion
- Coroutinen können ihren Ablauf unterbrechen und später wieder aufnehmen wobei sie ihren Status beibehalten.

Prozedur



Coroutinen



Coroutinen/Prozesse

- Anlegen der Coroutine mittels NEWCOROUTINE
- Braucht
  - Eine Prozedur die Coroutine implementier
  - Freien Arbeitsspeicherbereich
  - Eine COROUTINE Variable für späteren TRANSFER

```
VAR
   Main,Process1 : COROUTINE;
   WorkSpace1  : ARRAY[1..3000] OF WORD;

PROCEDURE SubProcess;
BEGIN … END;

NEWCOROUTINE(SubProcess,ADR(WorkSpace1),SIZE(WorkSpace1),Process1);
TRANSFER(Main, Process1);
```

Quellen Prozess; wird gesetzt

Target Prozess;

Beispiel Coroutinen

```
MODULE Corout;
FROM Terminal IMPORT WriteCard, WriteString, WriteLn;
FROM SYSTEM IMPORT WORD, ADR, TSIZE;
FROM COROUTINES IMPORT COROUTINE, NEWCOROUTINE, TRANSFER;
VAR  main, Process1, Process2 : COROUTINE;
     WorkSpace1, WorkSpace2   : ARRAY[1..3000] OF WORD;

PROCEDURE MainProcess;
VAR Index : CARDINAL;
BEGIN
   FOR Index := 1 TO 5 DO
      WriteString('This is loop');
      WriteCard(Index,2);
      IF Index > 2 THEN
         TRANSFER(Process1,Process2);
         WriteString(' and back to main loop');
      END;
      WriteLn;
   END;     (* of FOR loop *)
   WriteString('End of the MainProcess loop');
   WriteLn;
   TRANSFER(Process1,main);
END MainProcess;
```

… Beispiel Coroutinen

```
PROCEDURE SubProcess;
BEGIN
   LOOP
      WriteString(' in SubProcess');
      TRANSFER(Process2,Process1);
      WriteString(' back');
   END;
END SubProcess;

BEGIN   (* Main Module Body *)
   WriteString('Start the program....');
   NEWCOROUTINE(MainProcess,ADR(WorkSpace1),SIZE(WorkSpace1),Process1);
   NEWCOROUTINE(SubProcess,ADR(WorkSpace2),SIZE(WorkSpace2),Process2);
   TRANSFER(main,Process1);
   WriteString('End of the program');
   WriteLn;
END Corout.
```

Fragen

System Module

- System Module: SYSTEM, COROUTINES,...

- Keine Übersetzung dieser Module notwendig.
- Durch die Systemmodule werden bereitgestellt:
  - rechnerspezifischer Datentypen und Prozeduren,
  - Typtransfer-Funktionen und die Möglichkeit der Deklaration von Variablen mit festen Adressen.

- Durch SYSTEM-Modul bereitgestellte Datentypen:
  - WORD, ADDRESS, PROCESS.

- Durch SYSTEM-Modul bereitgestellte Prozeduren:
  - ADR, SIZE, TSIZE, NEWPROCESS, TRANSFER sowie weitere Objekte, abhängig von verwendeten Compiler.

Beispiel 1: Stack

```
DEFINITION MODULE Stack;
   TYPE Stack = RECORD
       top: INTEGER;
       value: ARRAY [0..999] OF INTEGER;
   END;
   PROCEDURE Clear (VAR s: Stack);
   PROCEDURE Push (VAR s: Stack; i: INTEGER);
   PROCEDURE Pop (VAR s: Stack; VAR i: INTEGER);
END Stack.
IMPLEMENTATION MODULE Stack;
    PROCEDURE Clear (VAR s: Stack);
    BEGIN
       s.top := 0;
    END Clear;
    PROCEDURE Push (VAR s: Stack; i: INTEGER);
    BEGIN
       IF s.top <= 999 THEN
       s.value[s.top] := i; INC(s.top);
       END;
    END Push;
    PROCEDURE Pop (VAR s: Stack; VAR i: INTEGER);
    BEGIN
       IF s.top > 0 THEN
       DEC(s.top);
       i := s.value[s.top];
    END; END Pop;
END Stack.
```

Beispiel 2: Zufallszahlen

```
DEFINITION MODULE Generator;
(* by R. Sutcliffe  revised 1993 04 06 *)
PROCEDURE RANDOM( ) : REAL;
(* Returns a random number 0 .. 1 *)
END Generator.

IMPLEMENTATION MODULE Generator;
(* by R. Sutcliffe  revised 1993 04 06 *)
VAR s1, s2: INTEGER;

PROCEDURE RANDOM(): REAL;
    CONST scale = 4.656613E-10;
    VAR Z, k: INTEGER;
    BEGIN
        k := s1 DIV 53668;
        s1 := 40014 * (s1 - k*53668) - k*12211;
        IF s1 < 0 THEN INC (s1, 2147483563) END(*IF*);
        k := s2 DIV 52774;
        s2 := 40692 * (s2 - k*52774) - k*3791;
        IF s2 < 0 THEN INC (s2, 2147483399) END(*IF*);
        Z := s1 - s2;
        IF Z < 1 THEN INC (Z,  2147483562) END(*IF*);
        RETURN scale*FLOAT(Z);
    END RANDOM;

BEGIN   (* initialize seed in body of module*)
  s1:=17; s2 = 113;
END Generator.
```

```
PROCEDURE Init;
 VAR dt : SysClock.DateTime;
     sv : INTEGER;

 BEGIN
     SysClock.GetClock(dt);
     sv := dt.fractions + (1000 *
       (dt.second + (60 *
        (dt.minute + (60 * dt.hour)))));
     s1 := sv;
     s2 := sv;

 END Init;
```

Beispiel 3: PrimZahlen

```
MODULE PrimZahlen;
FROM Terminal IMPORT WriteString, WriteInt, WriteLn;
CONST Max = 100; (* 2 < Primzahlen < 2*Max+1 *)
VAR Zahlen : ARRAY [1..Max] OF BOOLEAN;
    I,J    : INTEGER;
BEGIN
    WriteString("Primzahlen",0);
    WriteLn;
    FOR I:= 1 TO Max DO Zahlen[I]:= TRUE END;
    FOR I:= 3 TO (2*Max+1) DIV 3 BY 2 DO
           FOR J:= 3 TO (2*Max+1) DIV I BY 2 DO
                  Zahlen[I*J DIV 2] := FALSE
           END
    END;
    J:= -1;
    FOR I:=1 TO Max DO
           IF Zahlen[I] THEN
           J:= J+1;
           WriteInt(2*I+1,6);
           IF J MOD 10 = 9 THEN WriteLn END
           END
    END;
    WriteLn
END Primzahlen.
```

Module Terminal

```
DEFINITION MODULE Terminal;
PROCEDURE Read (VAR ch: CHAR);
PROCEDURE ReadLn (VAR s: ARRAY OF CHAR); (* stops at end-of-line *)
PROCEDURE BusyRead (VAR ch: CHAR);
PROCEDURE ReadAgain;
PROCEDURE Write (ch: CHAR);
PROCEDURE WriteString (s: ARRAY OF CHAR);
PROCEDURE WriteLn;
END Terminal.
```

MODULE SWholeIO und SRealIO;

```
DEFINITION MODULE SWholeIO;
PROCEDURE ReadInt (VAR int: INTEGER);
PROCEDURE WriteInt (int: INTEGER; width: CARDINAL);
PROCEDURE ReadCard (VAR card: CARDINAL);
PROCEDURE WriteCard (card: CARDINAL; width: CARDINAL);
END SWholeIO.
```

```
DEFINITION MODULE SRealIO;
ROCEDURE ReadReal (VAR real: REAL);
PROCEDURE WriteFloat (real: REAL; sigFigs: CARDINAL; width: CARDINAL);
PROCEDURE WriteEng (real: REAL; sigFigs: CARDINAL; width: CARDINAL);
PROCEDURE WriteFixed (real: REAL; place: INTEGER; width: CARDINAL);
PROCEDURE WriteReal (real: REAL; width: CARDINAL);
END SRealIO.
```

Set Standard Channel to Terminal

To use SWholeIO or SRealIO the Standard Channel has to be set as follow

```
FROM StdChans IMPORT SetOutChan;
FROM TermFile IMPORT ChanId, Open, read, write, echo, OpenResults;

VAR ch      : CHAR;
  term : ChanId;
  resOpen : OpenResults;

BEGIN
    (* Open new channel to the terminal in single character mode *)
    Open (term, write+read+echo, resOpen);
    SetOutChan(term);
```

Module SYSTEM

```
DEFINITION MODULE SYSTEM;
TYPE
  ADDRESS; WORD;
PROCEDURE ADR (anyObject: ARRAY OF WORD): ADDRESS;
PROCEDURE SIZE (anyObject: ARRAY OF WORD): CARDINAL;
   (* older versions--newer ones have it as a built-in *) PROCEDURE TSIZE (anyType
   [,optional variant list] ): CARDINAL;
PROCEDURE NEWPROCESS (nameOfProcess : PROC;
          workspace  : ADDRESS;
          sizeOfSpace   : CARDINAL;
          VAR newProc  : ADDRESS);
PROCEDURE TRANSFER  (VAR oldProcess : ADDRESS; VAR newProcess : ADDRESS);
PROCEDURE IOTRANSFER (VAR oldProcess : ADDRESS; VAR newProcess : ADDRESS);
(* Not all implement the latter *)
END SYSTEM.
```

Module Storage

```
DEFINITION MODULE Storage;
FROM SYSTEM IMPORT
  ADDRESS;
PROCEDURE ALLOCATE (VAR p: ADDRESS; size: CARDINAL);
PROCEDURE DEALLOCATE (VAR p: ADDRESS; size: CARDINAL);
PROCEDURE Available (size: CARDINAL): BOOLEAN;
END Storage.
```

Module RealMath

```
DEFINITION MODULE MathLib0;
CONST
   pi = 3.1415926536;
   e  = 2.7182818284;

PROCEDURE sqrt (x: REAL): REAL;
PROCEDURE exp (x: REAL): REAL;
PROCEDURE ln (x: REAL): REAL;
PROCEDURE sin (x: REAL): REAL;
PROCEDURE cos (x: REAL): REAL;
PROCEDURE arctan (x: REAL): REAL;
PROCEDURE real (x: INTEGER): REAL;
PROCEDURE entier (x: REAL): INTEGER;
END MathLib0.
```

Module Strings

```
DEFINITION MODULE Strings;
  TYPE
    STRING = ARRAY [0..80] OF CHAR;

PROCEDURE Assign (VAR source, dest: ARRAY  OF CHAR);
PROCEDURE Insert (substr: ARRAY  OF CHAR; VAR str: ARRAY  OF CHAR; index: CARDINAL);
PROCEDURE Delete (VAR str: ARRAY  OF CHAR; index: CARDINAL; len: CARDINAL);
PROCEDURE Pos (substr, str: ARRAY  OF CHAR): CARDINAL;
PROCEDURE Copy (str: ARRAY  OF CHAR; index: CARDINAL; len: CARDINAL;
          VAR result: ARRAY  OF CHAR);
PROCEDURE Concat (s1, s2: ARRAY  OF CHAR; VAR result: ARRAY  OF CHAR);
PROCEDURE Length (VAR str: ARRAY  OF CHAR): CARDINAL;
PROCEDURE CompareStr (s1, s2: ARRAY  OF CHAR): INTEGER;
           (* returns -1 if s1 < s2  0 if s1 = s2  1 if s1 > s2 *)
END Strings.
```