Inhalt der Vorlesung

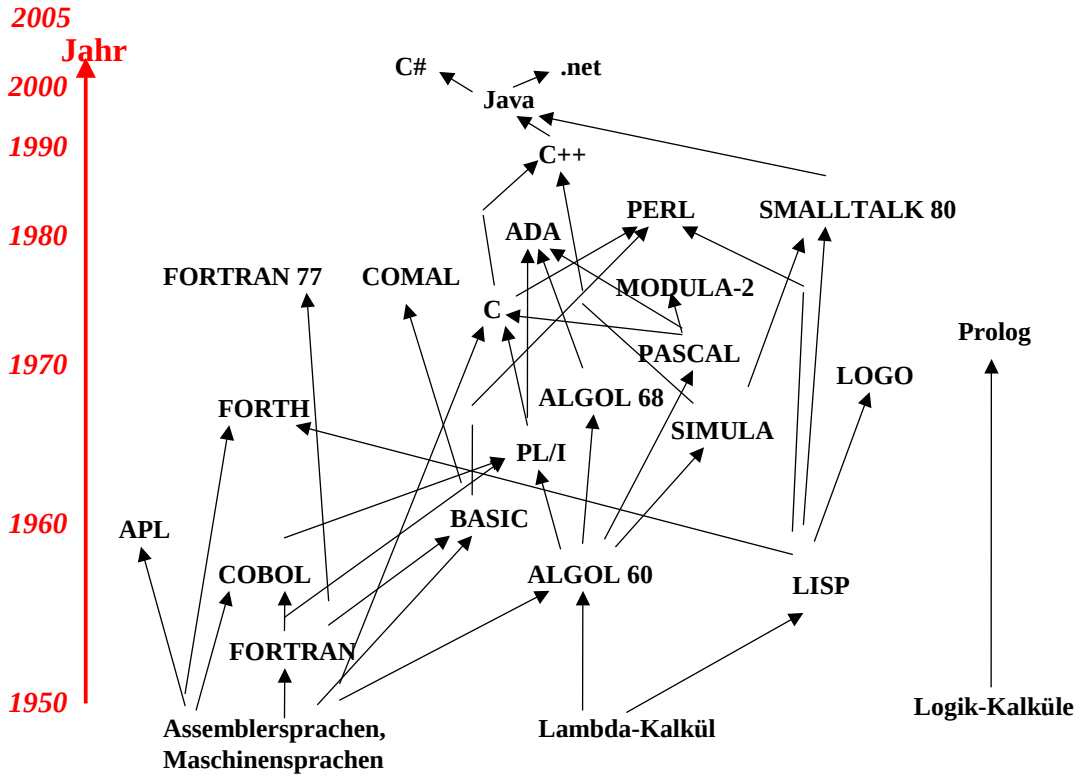
3. Die Konzepte einer Programmiersprache: Modula-2

Inhaltsübersicht

Software-Generation

Zeit	*Software*
Bis Ende der 50er (1.Generation)	Programmierung in **Maschinencode**, sehr einfache Betriebssysteme
Bis Ende der 60er (2.Generation)	Problemorientierte Programmiersprachen, **Mehrprogramm**-Betriebssysteme
Seit Mitte der 60er (3.Generation)	Dialogbetrieb, **Datenbanken**, strukturierte Programmierung
Seit Mitte der 70er (4.Generation)	Rechnerkommunikation, **verteilte** Systeme, Programmierumgebungen, graphische Benutzeroberflächen, **objektorientierte** Programmierung,
...In den 90ern (5.Generation)	Wissensverarbeitung, Parallelverarbeitung, automatisches Schließen, **WWW**, Multimedia, **ecommerce**,...

Software-Generation



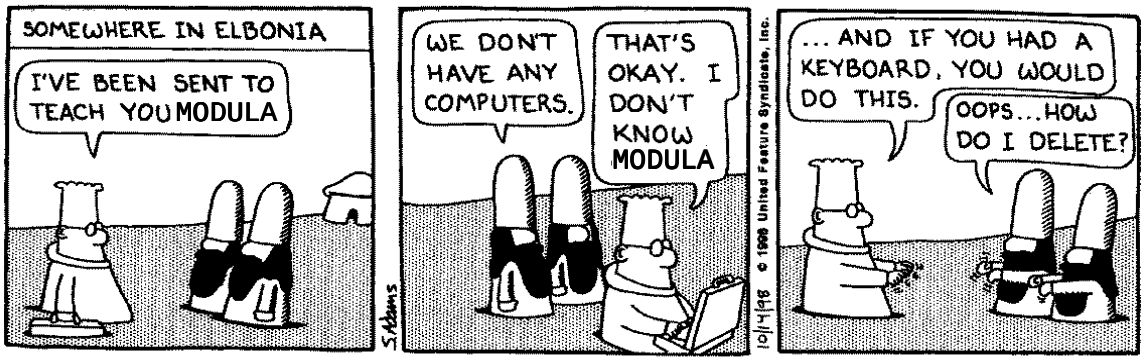
Software-Generation

Programmiersprachen können – gemäß ihrem **Paradigma** – in **verschiedene Kategorien** eingeteilt werden:

- **Imperative** Programmiersprachen:
  C, **Modula-2**, Pascal, ...
- **Funktionale** Programmiersprachen:
  LISP, ...
- **Prädikative** Programmiersprachen:
  Prolog, ...
- **Objektorientierte** Programmiersprachen:
  Smalltalk, Eiffel, C++, **Java**, ...

Motivation - Wieso Modula-2?

Wir haben die für die Vorlesung **Praktische Informatik I** die Programmiersprache **Modula-2** ausgewählt – **Wieso?**



Wieso Modula-2?

Vorteile von Modula-2:

- Modula-2 hat ein klares und schönes **Design**.
- Modula-2 ist sehr nützlich, um die Konzepte des **Programmierens** zu vermitteln.
- Modula-2 enthält alle relevanten Konzepte einer **modernen imperativen Programmiersprache** (im Gegensatz zu Cobol oder Fortran).
- Modula-2-Compiler sind (frei) **verfügbar**.

Wieso Modula-2?

Nachteile von Modula-2:

- Modula-2 ist relativ **unbekannt** und **nicht** die in der Industrie am meisten eingesetzte Programmiersprache.

ABER:

Wenn Sie Modula-2 kennen, werden Sie in der Lage sein,

- **gute** Programme zu entwickeln.
- und andere Programmiersprachen **schnell** zu erlernen.

Im letzten Teil der Vorlesung **Praktische Informatik I** werden wir uns mit der “aktuelleren” Programmiersprache **Java** befassen, die besonders für die **Programmierung im WWW** interessant ist.

Entwicklungsgeschichte

Modula-2 wurde von **Nikolaus Wirth** an der ETH Zürich entwickelt.

Die Sprache Modula-2 ist der **Nachfolger** der direkten Vorgänger **Pascal** (1970) und Modula (1975).

- In Modula-2 sollten alle Aspekte von Pascal vertreten sein. Außerdem sollte es um die wichtigen Konzepte der **Modularität** und des **Multiprogrammings (coroutines)** erweitert werden.
- Eine erste Implementierung wurde 1979 fertiggestellt. Compiler sind seit 1981 verfügbar.
- Für Modula-2 wird öfters auch nur der Begriff Modula verwendet.

Motivation

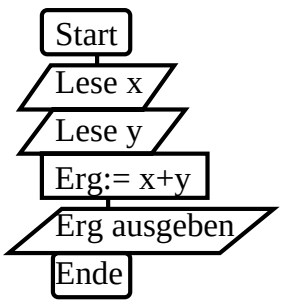
Programmieren Sie einen Taschenrechner

Programmierer

Finde einen Algorithmus für die gegebene Aufgabe

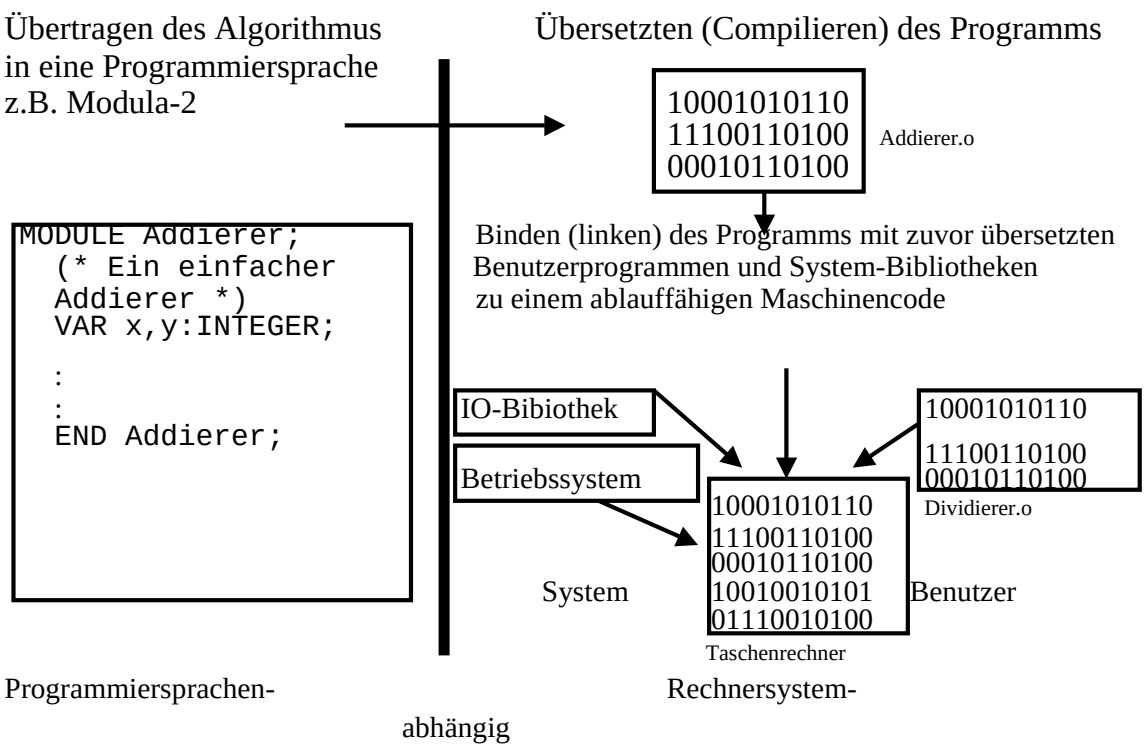
T1: Lese x
T2: Lese y
T3: Erg:=x+y
T4: Gebe Erg aus

oder



Dieser Entwurf ist **unabhängig** von Programmiersprachen und Rechnersystem

Motivation



Motivation

Aufgabe:

Anhand der Implementierung eines Taschenrechners zeigen wir:

- **Inkrementellen Entwurf**
- **Syntax von Modula-2 anhand von Beispielen**
- **Programmierung in Modula-2**

Taschenrechner Version 1

Das erste Modula-2 Programm:

Ausgabe einer Zeichenkette auf dem Bildschirm.

Zeichenketten werden im Programm mit Anführungszeichen gekennzeichnet.

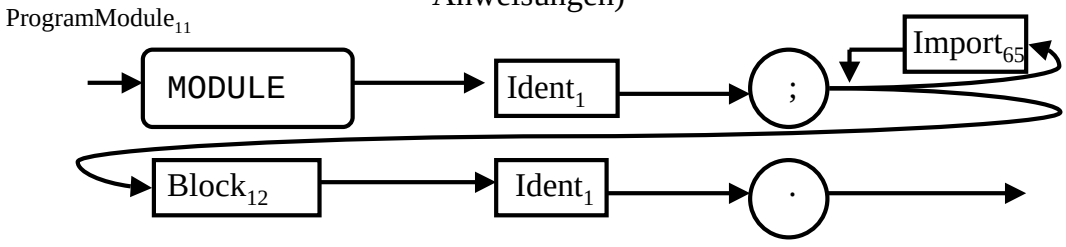
```
MODULE Taschenrechner;
FROM InOut IMPORT WriteString, WriteLn;
BEGIN
   WriteString ("Hallo, ich bin ein Taschenrechner.");
   WriteLn;
END Taschenrechner.
```

Beispiel eines Aufrufes von Taschenrechner:

Ausgabe: **`Hallo, ich bin ein Taschenrechner.`**

Programmstruktur

- Ein Hauptmerkmal von Modula-2 - im Gegensatz zu älteren Sprachen – ist das **Modul-Konzept**
- Module sind eigenständige, abgeschlossene Programmfragmente (später mehr)
- ein Programm besteht aus einem Hauptmodul („ProgramModule“) und evtl. weiteren Modulen
- ein Modula-2-Programm gliedert sich in
  - Module (Teil-Programme)
  - Prozeduren (kleinere funktionale Einheiten (Teile von Modulen))
  - Blöcke (logisch zusammenhängende Folge von Anweisungen)



Programmstruktur

- Syntax-Diagramme reichen nicht aus, um korrekte Programme zu schreiben.
- Es müssen noch Kontextbedingungen erfüllt sein.
  Zum Beispiel müssen die Bezeichner, die auf `MODULE` und das abschließende `END` folgen, gleich sein.

Beispiel:

Unzulässiges Programm

```
MODULE X; FROM InOut IMPORT (*PROC*) WriteLn;
BEGIN END Y.
```

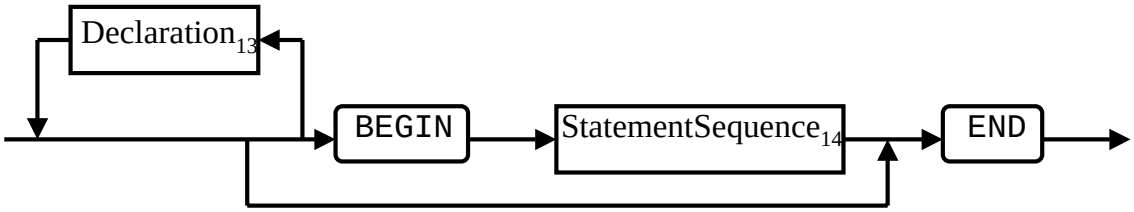
Syntaktisch korrektes Programm

```
MODULE Jubel;
FROM InOut IMPORT (*PROC*) WriteString, WriteLn;
BEGIN
    WriteSting("Informatik ist schoen!"); WriteLn;
END Jubel.
```


Block

- Ein Block besteht aus beliebig vielen *Deklarationen* und *Anweisungen* (Statements)
- Deklarationen und Anweisungen müssen durch `BEGIN` getrennt werden.

Block$_{12}$



Schlüsselwörter

- Modula-2 kennt einige Wörter mit fester Bedeutung, die sogenannten Schlüsselwörter (**Key Words)**
- Diese besitzen eine definierte, nicht änderbare Semantik (**Bedeutung**).
- Schlüsselwörter sind **reserviert** und dürfen nicht als Bezeichner für (benutzer)eigene Module, Prozeduren, etc. verwendet werden.
- Schlüsselwörter müssen immer **groß** geschrieben werden.

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

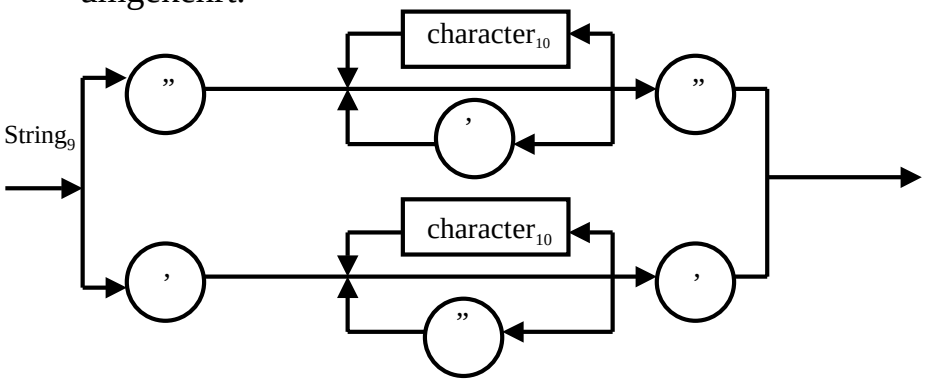

Zeichenketten

- Zeichenketten (String) sind beliebig lange Folge von Zeichen.
- Der Zeichenvorrat (Zeichensatz) hängt vom benutzten Rechner ab.
- Ein häufig benutzter Zeichensatz ist **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange). Er enthält Groß- und Kleinbuchstaben, Ziffern, Sonderzeichen und Steuerzeichen.

NUL 0	DLE 16	␣ 32	0 48	@ 64	P 80	@ 96	p 112
SOH 1	DC1 17	! 33	1 49	A 65	Q 81	a 97	q 113
STX 2	DC2 18	" 34	2 50	B 66	R 82	b 98	r 114
ETX 3	DC3 19	# 35	3 51	C 67	S 83	c 99	s 115
EOT 4	DC4 20	$ 36	4 52	D 68	T 84	d 100	t 116
ENQ 5	NAK 21	% 37	5 53	E 69	U 85	e 101	u 117
ACK 6	SYN 22	& 38	6 54	F 70	V 86	f 102	v 118
BEL 7	ETB 23	' 39	7 55	G 71	W 87	g 103	w 119
BS 8	CAN 24	(40	8 56	H 72	X 88	h 104	x 120
HT 9	EM 25	) 41	9 57	I 73	Y 89	i 105	y 121
LF 10	SUB 26	* 42	: 58	J 74	Z 90	j 106	z 122
VT 11	ESC 27	+ 43	; 59	K 75	[91	k 107	{ 123
FF 12	FS 28	, 44	< 60	L 76	\ 92	l 108	| 124
CR 13	GS 29	- 45	= 61	M 77	] 93	m 109	} 125
SO 14	RS 30	. 46	> 62	N 78	^ 94	n 110	~ 126
SI 15	US 31	/ 47	? 63	O 79	_ 95	o 111	DEL 127

Zeichenketten

- Die Begrenzer ' ' (einfache Hochkommata) oder " " (doppelte Hochkommata) zählen nicht zum String ("" hat Länge Null).
- Ein durch ' ' begrenzter String darf beliebige " enthalten und umgekehrt.



Beispiel für Modula-2-Zeichenketten:

- Gültig: 'Informatik I' oder "Uni Frankfurt" oder "Er sagte 'Hallo' "
- Ungültig: "Ungültige Zeichenketten!' oder "Auch "falsch""

Taschenrechner Version 2

Wie kann der Benutzer des Rechners Eingaben machen?

```
MODULE Taschenrechner;

FROM   InOut IMPORT ReadInt, WriteString, WriteLn;
VAR    x,y : INTEGER;

BEGIN
   WriteString ('Hallo, ich bin ein Taschenrechner.');
   WritelLn;
   WriteString('Eingabe x:'); ReadInt(x);
   WriteString('Eingabe y:'); ReadInt(y);
END Taschenrechner.
```

Beispiel eines Aufrufes von Taschenrechner:

Ausgabe: `Hallo, ich bin ein Taschenrechner.`
`Eingabe x:  Eingabe y:`

Eingabe: x y

Deklarationen

- Jedes Programm verarbeitet "Objekte", z.B. Zahlen, Texte, Zeichenketten, Bilder, etc.

- Objekte haben einen Typ (legt Wertebereich, Interpretationen und Verwendung fest).
  z.B. Integer, Real, Character, Boolean,...

- Alle benannten Objekte müssen deklariert (vereinbart) werden; Die Deklaration verbindet ein Objekt mit einer Typbezeichnung und einem Namen.

Deklarationen

- Modula-2 kennt sechs „elementare“ Typen, die folgende (reservierten) Bezeichner tragen:

```
INTEGER, CARDINAL, REAL,
BOOLEAN, CHAR, BITSET
```

- das Typkonzept von Modula-2 erlaubt es, eigene Typen den eingebauten Typen hinzuzufügen (durch eine Typdeklaration)

- der Compiler weiß aufgrund der Typangabe, wieviel Speicherplatz er für das Objekt reservieren muss, welche Operatoren auf das Objekt anwendbar sind, ob der Programmierer dieses Objekt immer typgerecht verwendet, etc. (schützt vor missbräuchlicher Verwendung)

Typen

`INTEGER`

- ganze Zahlen innerhalb des auf dem Rechner darstellbaren Wertebereichs
- 16-Bit-Rechner bieten Werte zwischen -32768 und 32767
- Operatoren: `+,-,*,DIV,MOD,=,<>,<,>,<=,>=`

`CARDINAL`

- positive ganze Zahlen mit rechnerabhängigem Wertebereich
- 16-Bit-Rechner bieten Werte zwischen 0 und 65535
- Operatoren analog zu `INTEGER`

Typen

`REAL`

- gebrochene Zahlen (Dezimalbrüche) mit einem vom Rechnertyp abhängigen Wertebereich
- Operatoren: `+,-,*,/,=,<>,<,>,<=,>=`
- unvollkommene Repräsentation im Rechner führt zu Rechenungenauigkeiten.
  z.B. (1/3)*3 ergibt den Wert 0.9999999

`BOOLEAN`

- Wahrheitswerte (boolesche Werte)
- Wertebereich: `TRUE` und `FALSE`
- Operatoren: `AND, OR, NOT, <, >`

Typen

CHAR

- Zeichen des auf dem jeweiligen Rechner verfügbaren Zeichensatzes
- Jedem Zeichen ist eine Ordinalzahl zugeordnet, die beim Vergleichen von CHARs verwendet wird
- Die Buchstaben und Ziffern haben aufeinanderfolgende Ordinalzahlen (vgl. ASCII-Tabelle).

Beispiel.: Die Frage „Ist ein Objekt c des Types CHAR eine Ziffer?“ kann in Modula-2 formuliert werden durch:

```
( c>= "0" ) AND (c <= "9")
```


Benutzerdefinierte Typen

Typen sind durch den Benutzer definierbar:

- Der Programmierer kann eigene Datentypen definieren und dadurch die vorhandenen Typen erweitern
- Der Programmierer kann somit Programme lesbarer und änderungsfreundlicher machen.
- **Aufzählung** (**Enumeration**) und **Unterbereich** (**Subrange**) sind zwei einfache Möglichkeiten (später werden noch weitere Möglichkeiten vorgestellt).

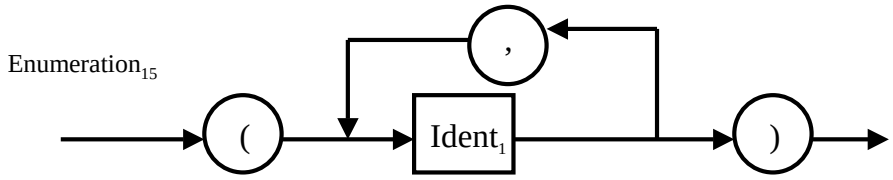
Benutzerdefinierte Typen

Enumeration (Aufzählungstyp)

- stellt einen neuen Typ zur Verfügung, der nur Elemente aus der angegebenen Aufzählung enthalten kann.

Beispiel.: Fische = (forelle, karpfen, hecht, fugu, hai)

- Enumerationskonstanten werden beginnend mit 0 aufsteigend nummeriert. Es gilt z.B. karpfen < fugu



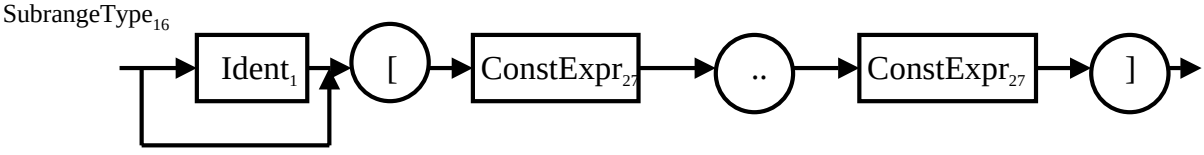
Benutzerdefinierte Typen

Subrange (Unterbereichstyp)

Definiert einen neuen Typ, der aus einem angegebenen Bereich eines Basistypen besteht.

```
[0 .. 9]
['a' .. 'z']
Fische  [forelle .. hecht]
INTEGER [294 .. 2411]
```

- Angaben von Minimal- und Maximalwert
- Zulässig sind alle Basistypen, die mit Ordinalzahlen versehen sind.
- Der Basistyp kann durch vorgestellten qualifizierten Bezeichner festgelegt werden.



Konstantendeklaration

- Eine **Konstante** ist ein Objekt der Verarbeitung, dessen Wert sich während der Ausführung **nicht** ändern kann.
- Eine Konstantendeklaration darf nur aus (schon bekannten) Konstanten bestehen.

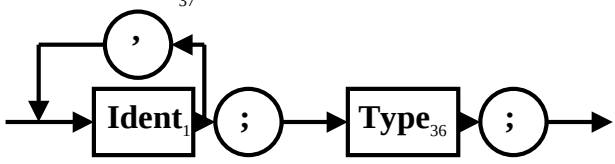
Beispiel:

```
CONST
   wordsize = 16;
   Bytes = wordsize DIV 8;
   Label = "Giftig!"
   minus1 = -1;
   Pi = 3.141;
```


Variablendeklaration

- Eine Variable ist ein Objekt der Verarbeitung, dessen Wert sich während der Ausführung ändern kann
- Variablen müssen vor ihrer Benutzung deklariert werden, sie besitzen einen **Namen** (Identifier) und einen **Typ**

VariableDeclaration$_{37}$



Beispiel:

```
VAR
   i: INTEGER;
   x,y: REAL;
   mitte : Koordinate;
   name: ARRAY [1..29] OF CHAR;
   wanda: Fisch;
```

Taschenrechner Version 3

Wie wird eine Variable wieder ausgegeben ?

```
Module Taschenrechner;
(* Dies ist das Taschenrechnerprogramm Version 3 *)

FROM InOut IMPORT WriteInt,ReadInt,WriteString,WriteLn;
VAR   x,y : INTEGER;

BEGIN
   WriteString ('Hallo, ich bin ein Taschenrechner.');
   WritelLn;
   WriteString('Eingabe x:'); ReadInt(x);
   WriteString('Eingabe y:'); ReadInt(y);
   WriteString('x enthält jetzt:'); WriteInt(x);
   WriteLn;
   WriteString('y enthält jetzt:'); WriteInt(y);
END Taschenrechner.
```


Kommentare

- Kommentare sollen den Programmtext **verständlich** machen.
- Gute Kommentierung ist **extrem wichtig** für die Pflege und Anpassung von Programmen.
- Kommentare im Programmtext werden vom Compiler **ignoriert**
- Kommentare können auch dazu benutzt werden, um Teile eines Programms ***auszukommentieren***.
- Kommentare in Modula-2 werden durch **(* und *)** geklammert (auch geschachtelt möglich).

Beispiel:

Gültig: `(*  Programm : Test  *)`

Ungültig: `(* Author  : Heinz Hinz )`

Schlecht sind zum Beispiel kryptische Abkürzungen:

```
VAR x: INTEGER; (* x i.e. Int.Var. *)
```