# Ein- und Ausgabe

## Übersicht

- Wie werden Schriftzeichen kodiert?
- Ein- und Ausgabe von ASCII-Codes
  - Eingabe: get/1, get0/1
  - Ausgabe: put/1, nl/0, tab/1
  - Konvertierung: name/2, atom_codes/2, number_codes/2
  - Zeichenketten
- Ein- und Ausgabe von Termen
  - write/1, read/1, write_canonical/1
- Umlenken von Ein- und Ausgabe in Dateien
  - tell/1, telling/1, told/0
  - see/1, seeing/1, seen/0
- Dateienden und -Verarbeitung



# Buchstaben als Zahlen: Kodierung

## Buchstaben können als Zahlen angesehen werden.

- Eine *Kodierung* legt fest, welcher Buchstabe mit welcher Zahl gemeint ist.

| *Willkürliche Zuordnung 1* | *Willkürliche Zuordnung 2* |
|---|---|
| 1 = A | 65 = A |
| 2 = B | 66 = B |
| … | … |

- Die Zahlen selbst wurden 'traditionell' durch 8-stellige 0/1-Folgen (*byte*) dargestellt. D.H. 256 mögliche unterschiedliche Werte

| *Willkürliche Zuordnung 1* | *Willkürliche Zuordnung 2* |
|---|---|
| 00000001 = A | 01000001 = A |
| 00000010 = B | 01000010 = B |
| … | … |



# Kodierungsstandards

## Verschiedene Standards/Konventionen

- ***A**merican **S**tandard **C**ode for **I**nformation **I**nterchange* (ASCII)
  - anderer Name: *International Alphabet* 5 (IA5)
  - regelt Codes fürs englische Alphabet (A – Z; a – z; 0 – 9) und einige Sonderzeichen wie @, {, /, (, %, [.
- **ISO 8859-1** erweitert ASCII um Codes für die Schriftzeichen der meisten westeuropäischen Sprachen, z.B. ä, ß, É, Ò
  - UNIX- und WIN-Systeme verwenden oft ISO-8859-1 (ANSI). MacOS und DOS nicht.
- Dutzende andere Konventionen

## Probleme

- Manche Konventionen widersprechen sich (MacOS vs. WIN)
- Nur ASCII ist wirklich weit verbreitet, umfasst aber wenig Zeichen



# ASCII-Codetabelle (Zeichensatz)

## Ausschnitt aus den 128 Zeichen der ASCII-Tabelle

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 10/13 | *neue Zeile* | 47 | / | 63 | ? | 79 | O | 95 | _ | 111 | o |
| 32 | *Leerschlag* | 48 | 0 | 64 | @ | 80 | P | 96 | ` | 112 | p |
| 33 | ! | 49 | 1 | 65 | A | 81 | Q | 97 | a | 113 | q |
| 34 | " | 50 | 2 | 66 | B | 82 | R | 98 | b | 114 | r |
| 35 | # | 51 | 3 | 67 | C | 83 | S | 99 | c | 115 | s |
| 36 | $ | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t |
| 37 | % | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u |
| 38 | & | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v |
| 39 | ' | 55 | 7 | 71 | G | 87 | W | 103 | g | 119 | w |
| 40 | ( | 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x |
| 41 | ) | 57 | 9 | 73 | I | 89 | Y | 105 | i | 121 | y |
| 42 | * | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z |
| 43 | + | 59 | ; | 75 | K | 91 | [ | 107 | k | 123 | { |
| 44 | , | 60 | < | 76 | L | 92 | \ | 108 | l | 124 | \| |
| 45 | - | 61 | = | 77 | M | 93 | ] | 109 | m | 125 | } |
| 46 | . | 62 | > | 78 | N | 94 | ^ | 110 | n | 126 | ~ |

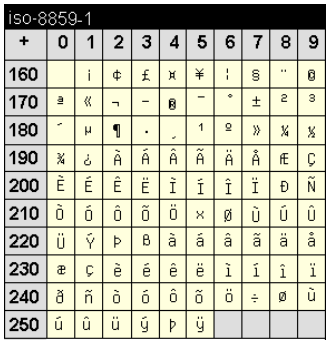
# Ein Ansatz zur Vereinheitlichung

**Unicode Version 3.0.1 (Dezember 2000)**

- Prinzip: Eine eindeutige Zahl für jedes Zeichen!
- Codes für alle gegenwärtig verwendeten Schriftzeichen (*glyphs*) und Symbole in (fast) allen Sprachen der Welt (49'194 Einträge)
- Codes für Zeichen einiger ausgestorbene Sprachen
- UTF-16 Kodierung mit 16-stelligen 0/1-Folgen (Zahlen von 0 bis 65535. D.H. maximal 65536 verschiedene Zeichen)
  - Konform zur ISO/IEC-Normierung 10646
  - Heute: Unterstützung durch Java, Windows NT, MacOS 8, …
  - Nahe Zukunft: WWW-Dokumente in Unicode (ab HTML 4)
  - UTF-32 Kodierung erlaubt sogar 32-stellige 0/1-Folgen
- Codetabellen und Infos unter `http://www.unicode.org`



# Ausschnitte Unicode-Codetabellen



| iso-8859-1 | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| + | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 160 | | ¡ | ¢ | £ | ¤ | ¥ | ¦ | § | ¨ | © |
| 170 | ª | « | ¬ | - | ® | ¯ | ° | ± | ² | ³ |
| 180 | ´ | µ | ¶ | · | ¸ | ¹ | º | » | ¼ | ½ |
| 190 | ¾ | ¿ | À | Á | Â | Ã | Ä | Å | Æ | Ç |
| 200 | È | É | Ê | Ë | Ì | Í | Î | Ï | Ð | Ñ |
| 210 | Ò | Ó | Ô | Õ | Ö | × | Ø | Ù | Ú | Û |
| 220 | Ü | Ý | Þ | ß | à | á | â | ã | ä | å |
| 230 | æ | ç | è | é | ê | ë | ì | í | î | ï |
| 240 | ð | ñ | ò | ó | ô | õ | ö | ÷ | ø | ù |
| 250 | ú | û | ü | ý | þ | ÿ | | | | |

Unicode enthält ISO-Latin 8859-1 zwischen 0 und 255

Exotischer Code



# Zeichen ausgeben: put/1

- **put/1** gibt ein einzelnes Zeichen aus. Das Argument ist der ASCII-Code des Zeichens.

```
?- put(72), put(97), put(108), put(108), put(111).
Hallo
```

**Allerdings definiert ASCII keine Codes für die Zeichen exotischer Sprachen.**

- Deutsch ist wegen Ä, Ö, Ü, ß etc. eine exotische Sprache
- Ergebnis von put(138) oder put(5000): nicht normiert!



# Druckbare Zeichen einlesen: get/1

- **get/1** wartet, bis der Benutzer ein einzelnes druckbares Zeichen auf der Tastatur eingibt.
  - Danach unifiziert das Argument von get/1 mit dem ASCII-Code des eingegebenen Zeichens.

```
?- get(X).
|: A
X = 65 ?
yes
```

Benutzer hat A eingegeben

| | |
|---|---|
| 63 | ? |
| 64 | @ |
| 65 | A |
| 66 | B |
| 67 | C |
| 68 | D |
| 69 | E |
| 70 | F |
| 71 | G |
| 72 | H |

Als **nicht-druckbar** gelten die ASCII-Zeichen von 0-32 und 127. Darunter sind Zeilenende (10/13), Tabulator (9) und Leerzeichen (32).

# Beliebige Zeichen einlesen: get0/1

- **get0/1** wartet, bis der Benutzer ein *beliebiges* Zeichen auf der Tastatur eingibt. (Eingabe durch Return!)
  - Danach unifiziert das Argument von get0/1 mit dem ASCII-Code des eingegebenen Zeichens.

```
?- get0(32).
yes
```

Benutzer hat Leerschlag (*space*) eingegeben!

| | |
|---|---|
| 32 | *Leerschlag* |
| 33 | ! |
| 34 | " |
| 35 | # |

Benutzer hat anderes Zeichen als Leerschlag eingegeben!

```
?- get0(32).
no
```



# Zeilenende nl/0 und Leerzeichen tab/1

- **Zeilenenden** werden auf unterschiedlichen Betriebssystemen durch unterschiedliche ASCII-Codes repräsentiert.

| Betriebssystem | ASCII |
|---|---|
| UNIX | 10 |
| MacOS | 13 |
| DOS/WIN | 10 + 13 |

10 = Zeilenvorschub (*linefeed*)
13 = Wagenrücklauf (*carriage return*)

  - **nl/0** gibt betriebsystemabhängig die richtigen ASCII-Codes für Zeilenende aus!
- **Mehrere Leerzeichen** werden gerne mit **tab/1** ausgegeben
  - Das Argument gibt die Anzahl der auszugebenden Leerzeichen an.

```
?- tab(1+2), put(33).
   !
yes
```



# Wie beweist Prolog Ein- und Ausgabe?

**Eingabeprädikate** sind bewiesen, wenn eine entsprechende Eingabe erfolgt ist.

- *Interaktive Eingabe "blockiert" den Prolog-Interpreter bis Input erfolgt.*
  - Bei Backtracking werden allfällige Variablenbindungen rückgängig gemacht, aber es erfolgt keine weitere Eingabeaufforderung!

**Ausgabeprädikate** sind bewiesen, wenn eine entsprechende Ausgabe als Seiteneffekt erfolgt ist.

- *Die Ausgabe selbst hat auf den Beweis keinen Einfluss.*
  - Bei Backtracking bleibt der Seiteneffekt (die Ausgabe) bestehen, kein Backtracking!

**Ein- und Ausgabe gelingen höchstens 1-Mal!**



# ASCII-Codes und atomare Terme

- **Das eingebaute Prädikat name/2**
  - gibt den Namen eines nicht-variablen atomaren Terms als Liste von ASCII-Codes zurück

```
?- name(bla, L).
L = [98,108,97]
```

```
?- name(27, L).
L = [50,55]
```

*Modus: name(+Atomar, ?Liste)*

  - oder erzeugt umgekehrt einen atomaren Term aus einer Liste von ASCII-Codes

```
?- name(A, [98,108,97]).
A = bla
```

*Modus: name(?Atomar, +Liste)*

# Das Problem mit name/2…

**Zahl oder Atom?**

- Falls die ASCII-Code-Liste eine Prolog-Zahl beschreibt, wird sie **immer nur** als Zahl instantiiert.

```
?- name(N, [50,55]).
N = 27 ;
no
```

```
?- name('27', [50,55]).
yes
```

Anstelle von name/2 sollten die konsistenten ISO-Prolog-Prädikate **atom_codes/2** und **number_codes/2** verwendet werden, die eine ASCII-Liste konsequent in Atome oder Zahlen umsetzen.

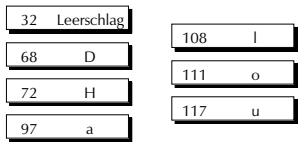
- Leider verwendet SICStus Prolog in älteren Versionen anstelle von atom_codes/2 atom_chars/2 und anstelle von number_codes/2 number_chars/2 …



# ASCII-Codes als Zeichenketten

**Eine beliebige Zeichenkette (*string*), die zwischen zwei " (doppelte Hochkommata) eingeschlossen ist, wird als Liste der ASCII-Codes dargestellt.**

```
?- Kette = "Hallo Du".
Kette = [72,97,108,108,111,32,68,117]
```






# Ein-/Ausgabe von Prolog-Termen

**Prolog hat vordefinierte Ein-/Ausgabe-Prädikate für Prolog-Terme**

- Vorteil: Komplexe Ausdrücke müssen nicht als Einzelzeichen eingelesen und mühsam zusammengesetzt werden
- Nachteil: Die Prolog-Term-Syntax muss beachtet werden
  - *Jeder Term muss bei der Eingabe mit einem Punkt beendet werden!*

**Interaktion kann mit der Aussenwelt nach einem einfachen Muster erfolgen.**

```
interaktion :-
    read(Input),
    verarbeite(Input, Output),
    write(Output).
```



# Terme einlesen: read/1

- **read/1** liest einen Term ein.

```
?- read(Eingabe), write(Eingabe).
|: bla.
bla
Eingabe = bla ? ;
no
```

*Eingabe des Benutzers*

*Ausgabe von Prolog*

- **Beachte:** Der Punkt beendet den Term, gehört aber selbst nicht dazu!

```
?- read(Eingabe), write(Eingabe).
|:bla bla.
{SYNTAX ERROR...
```

- **Beachte:** Die Syntaxregeln für Prolog-Terme müssen beachtet werden!

## Termausgabe: write/1, write_canonical/1

- **write/1** gibt einen einzelnen Term aus.

```
?- write(1 + 2 =:= 3 -0).
1+2=:=3-0
?- write([bla,bli,blu]).
[bla,bli,blu]
```

  - write/1 respektiert die beim Aufruf definierten Operatoren und Spezialsyntax.

- **write_canonical/1** ignoriert Spezialsyntax und Operatoren

```
?- write_canonical([bla,bli,blu] = 2).
=('.'(bla,'.'(bli,'.'(blu,[]))),2)
```



## Schreiben in eine Datei

- **tell/1 leitet die Ausgabe der Prädikate**
  - put/1
  - nl/0, tab/1
  - write/1, write_canonical/1

  **in eine Datei um.**

```
?- tell(hans),
   write(hallo), nl,
   write(du), nl,
   told.
```


- **told/0**
  - beendet die Umleitung (schliesst die Datei!)
  - sorgt dafür, dass zukünftige Ausgaben wieder auf dem Bildschirm erscheinen.




## Schreiben in eine Datei

- **telling/1** gibt an, in welche Datei die Ausgabe zur Zeit gerade geleitet wird.
  - `user` steht für den Bildschirm, der als abstrakte Datei betrachtet wird.

```
?- telling(Zuerst),
   tell(hans), write(hallo),
   telling(Mitte),
   told,
   telling(Zuletzt).
Zuerst = user,
Mitte = hans,
Zuletzt = user
```



## Lesen aus einer Datei

- **see/1 nimmt die Eingabe für die Prädikate**
  - get/1, get0/1
  - read/1

  **aus einer Datei.** (öffnet die Datei!)


- **seen/0**
  - beendet die Umleitung (schliesst die Datei!)
  - sorgt dafür, dass zukünftige Eingaben wieder vom Benutzer abgefragt werden.

```
?- see(hans),
   get0(_),
   get0(Y), put(Y), nl,
   seen.
a
```

# Lesen aus einer Datei

- **seeing/1** gibt an, aus welcher Datei die Eingabe zur Zeit gerade genommen wird.
  - `user` steht für die Tastatur, die als abstrakte Datei betrachtet wird.

```
?- seeing(Zuerst),
   see(hans),
   seeing(Mitte),
   seen,
   seeing(Zuletzt).
Zuerst = user,
Mitte = hans,
Zuletzt = user
```



# Das Dateiende

- **Wie kann beim Einlesen das Erreichen des Dateiendes erkannt werden?**

  **Dateiende wird als spezielles Element repräsentiert:**
  - **Eingabe mit ASCII-Kodes**
    - get/1, get0/1 liefern die Zahl -1 zurück.
  - **Eingabe mit Termen**
    - read/1 liefert das Atom end_of_file zurück.

- **Nachfrage: Wie wird beim Ausgeben das Dateiende herausgeschrieben?**
  - Prolog macht das automatisch beim Beweis von told/0.



# Verarbeitung eines Dateiinhalts

**Schema für Verarbeiten einer Datei mit Prolog:**

```
process_file(F) :-
  see(F),                % Oeffne Datei
  repeat,
    read(T),             % Term einlesen
    process_term(T),     % Term verarbeiten
    T == end_of_file,    % Fertig, falls
  !,                     % Dateiende
  seen.                  % Schliesse Datei
```

- repeat/0 ist ein eingebautes Prädikat, das beliebig oft gelingt.

```
repeat.
repeat :- repeat.
```

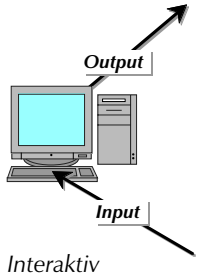


# Wer ist der user?

**Die abstrakten Dateien "user"**
- liefern Input von der Tastatur
- schreiben Output auf den Bildschirm

**im interaktiven Betrieb!**

***Aber:*** *Der Input kann auch von einem andern Programm kommen und an ein anderes ausgegeben werden!*




*Interaktiv*


*Prozesskommunikation*