2.4 Basistypen

2.4.1 Boolesche Ausdrücke

Ausdrücke sind strukturell definiert. Unterschied in der Repräsentation:
– die Konstanten sind `true` und `false`,
– die vordefinierten Konstruktoren sind `not`, `andalso` und `orelse`.

Achtung: `orelse` und `andalso` sind nicht strikt im 2. Argument (siehe Fallunterscheidung). Die Wahrheitswerte werden in ML ebenfalls durch `true` und `false` repräsentiert.

Beispiele:
```
–   (true);
>   true : bool
–   (true andalso false);
>   false : bool
–   ((not true) orelse (true andalso false));
>   false : bool
```

Klammerungsregeln (Präzedenzen) in ML:
– Die äußeren Klammern dürfen weggelassen werden.
– `not` bindet stärker als `andalso` bindet stärker als `orelse`.
– Überflüssige (grammatikalisch richtige) Klammern sind erlaubt.
– t = s mit einem Ausdruck von Typ `bool`, wenn t und s den gleichen Typ haben.

2.4.2 Ganze Zahlen [Wik, Kap.2.1]

Ausdrücke sind strukturell induktiv definiert (ähnlich wie Boolesche Ausdrücke) mit den Operatoren ~ (unäres minus), + (Addition), – (Subtraktion), * (Multiplikation), `div` (ganzzahlige Division) und `mod` (Restoperator).

ML hat 10 Präzedenzen. Operatoren mit höherer Präzedenzzahl binden stärker.

Operator	*	div	mod	+	–
Präzedenz	7	7	7	6	6

Bei gleicher Präzedenz wird nach links gebunden.

Beispiele:
```
–   3+5*7 = 3+(5*7);
>   true : bool
–   3+5*7 = (3+5)*7;
>   false : bool
–   3 div 5 + 7 = (3 div 5)+7;
```

```
>   true : bool
–   13 div 5 mod 2 = (13 div 5) mod 2;
>   true : bool
```

Division durch Null ist nicht definiert und ergibt eine Fehlermeldung, die angibt, welcher Operator den Fehler verursacht hat.

Beispiele:

```
–   17 div 0;
>   Failure: div
–   18 div (18 mod 2);
>   Failure: div
–   18 mod (3 div 5);
>   Failure: mod
```

Vergleichsoperatoren sind binäre Funktionen von den (ganzen oder Gleitpunkt–) Zahlen in den Bereich der Wahrheitswerte. D.h. sie sind vom Typ
`(int * int) -> bool` oder `(real * real) -> bool`.

Die vordefinierten ML Vergleichsoperatoren sind = ("gleich"), <> ("ungleich"), < ("kleiner"), <= ("kleiner oder gleich"), > ("größer") und >= ("größer oder gleich"). Ihre Präzedenz ist 4 (vgl. [Reade, S.16]).

Beispiele:

```
–   3+4 = 4+3;
>   true : bool
–   3+4 = 7 div 1;
>   true : bool
```

Typcheck: ML überprüft die korrekte Anwendung von Operatoren (Funktionen). Falsche Anwendungen entstehen durch Typfehler.

Beispiel:

```
–   3 = true;
Type clash  in:  (3 = true)
Looking  for a:  int
I have found a:  bool
```

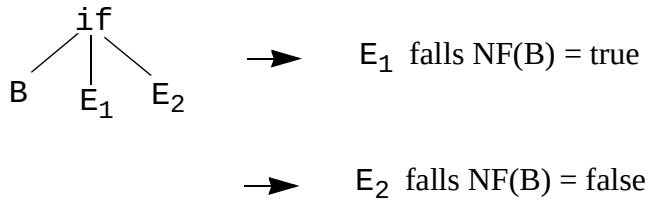
Standardfunktionen:
ML stellt eine Reihe von vordefinierten Funktionen bereit.

Beispiele: (für Standardfunktionen auf den ganzen Zahlen)

`abs:int -> int`	Absolutbetrag
(auch `real -> real`	"overloading")
`Int.max:int * int . int`	Maximum
`Int.min:int * int . int`	Minimum

Fallunterscheidung:
`if x then y else z;`

Auswertung von if-then-else
`if B then` E_1 `else` E_2`;`



Auswertung: Wenn der erste Parameter den Wahrheitswert `true` hat, dann hat der ganze `if` Ausdruck den Wert des zweiten Parameters, anderenfalls den des dritten Parameters.

Merke: `if` unterscheidet sich von allen anderen Funktionen in ML dadurch, daß der zweite oder dritte Parameter nicht unbedingt in jedem Aufruf definiert sein müssen. Eine Funktion, für die alle Parameter in jedem Aufruf definiert sein müssen, heißt **strikt**. `if` ist nicht strikt im zweiten und dritten Parameter.

Beispiele:

```
– if true then 0 else 1 div 0;
> 0 : int
– if false then 0 else 1 div 0;
Failure: div
– if true then 1 else 2.0;
Type clash  in:  (if true then 1 else 2.0)
Looking  for a:  int
I have found a:  real
```

2.4.3 Reelle Zahlen [Wik, Kap.2.2-2.3]

Reelle Konstanten **müssen** enthalten:
– entweder einen Dezimalpunkt
– oder einen Exponenten.

Beispiele:

```
– val pi = 3.14159265;          (eine Approximation von π)
> val pi = 3.14159265 : real
– 123.456E7;
> 1234560000.0 : real
– 123.456E20;
> 1.23456e22 : real
– 1E20;
> 1.0e20 : real
```

Konversion zwischen ganzen und reellen Zahlen **muß** explizit geschehen:

```
– pi*2;
Type clash  in:  (pi * 2)
looking  for a:  real
I have found a:  int
```

Division für `real` schreibt man `/`, dafür gibt es Konversionsfunktionen:

```
– pi * real(2);
> 6.263163 : real
```

Die Konversion von `int` nach `real` ist injektiv, die Konversion von `real` nach `int` aber nicht:

```
– floor(1.5); floor(1.6);
> 1 : int
> 1 : int
```

Eine reelle Zahl besteht aus
- einer **Mantisse** (einer endlichen Ziffernfolge)
- einem **Exponenten** (mit E gekennzeichnet)
- einem **Vorzeichen**.

Zur Darstellung reeller Zahlen als Gleitpunktzahlen lesen Sie bitte [Wik, S.30–31].
Weitere Funktion auf reellen Zahlen, z.B. Math.sin etc. Auf reellen Zahlen gibt es keine Gleichheit.

2.4.4 Strings [Wik, Kap.7]

Einzelne Buchstaben (characters) stellen einen Basistyp dar. In ML werden sie als einelementige Strings dargestellt.

```
- #"a";
> #"a": char
– size "hello"; size "a";
> 5 : int
> 1 : int
– "hello" ^ "there";                (Konkatenation)
> "hellothere" : string
– "hello\                           (Strings können mehrere
= \there";                           Zeilen lang sein)
> "hellothere" : string
```

Für die Textverarbeitung ist die Buchstabendarstellung im ASCII Code wichtig. Der ASCII Code definiert eine Abbildung von Schriftzeichen zu den ganzen Zahlen. In ML ist diese Funktion vordefiniert und heißt `ord`. Die Umkehrfunktion von `ord` ist `chr`.

```
– ord;
> fn: char–> int
– chr;
> fn: int –> char
- str;
```

```
> str: char -> string
```

Der ASCII Zeichensatz (oktal):

Steuerzeichen:

0	NUL	Null	20	DLE	Data link escape
1	SOH	Start of heading	21	DC1	Device control 1
2	STX	Start of text	22	DC2	Device control 2
3	ETX	End of text	23	DC3	Device control 3
4	EOT	End of transmission	24	DC4	Device control 4
5	ENQ	Enquery	25	NAK	Negative acknowledge
6	ACK	Acknowledge	26	SYN	Synchronous idle
7	BEL	Bell	27	ETB	End of transmission block
10	BS	Backspace	30	CAN	Cancel
11	HT	Horizontal tab (`\t`)	31	EM	End of medium
12	LF	Line feed (`\n`)	32	SUB	Substitute
13	VT	Vertical tab	33	ESC	Escape
14	FF	Form feed	34	FS	File separator
15	CR	Carriage return	35	GS	Group separator
16	SO	Shift out	36	RS	Record separator
17	SI	Shift in	37	US	Unit separator

Druckbare Zeichen:

40	(Space)	60	0	100	@	120	P	140		160	p
41	!	61	1	101	A	121	Q	141	a	161	q
42	"	62	2	102	B	122	R	142	b	162	r
43	#	63	3	103	C	123	S	143	c	163	s
44	$	64	4	104	D	124	T	144	d	164	t
45	%	65	5	105	E	125	U	145	e	165	u
46	&	66	6	106	F	126	V	146	f	166	v

47	’	67	7	107	G	127	W	147	g	167	w
50	(	70	8	110	H	130	X	150	h	170	x
51	)	71	9	111	I	131	Y	151	i	171	y
52	*	72	:	112	J	132	Z	152	j	172	z
53	+	73	;	113	K	133	[	153	k	173	{
54	,	74	<	114	L	134	\	154	l	174	|
55	–	75	=	115	M	135	]	155	m	175	}
56	.	76	>	116	N	136	^	156	n	176	~
57	/	77	?	117	O	137	_	157	o	177	(Delete)

Kontrollzeichen können mit ihrem dreistelligen dezimalen ASCII Code (escape sequence) spezifiziert werden (mit einem Präfix–Backslash). Manche Kontrollzeichen haben einen Namen.

```
– "\010"; "\n";
> "
" : string
> "
" : string
```

Es gibt eine einfache Korrespondenz zwischen Strings und Zeichenlisten. Die Funktion `explode` wandelt einen String in die Liste der in ihm enthaltenen Zeichen um, die Funktion `implode` formt aus einer Liste von Strings einen String.

```
– explode;
> fn : string -> (string list)
– implode;
> fn : (string list) -> string
– val x = explode "hello";
> val x = ["h","e","l","l","o"] : string list
– implode x;
> "hello" : string
```

2.4.5 Unit

Der Datentyp `unit` hat genau ein Element: `()`.

```
– ();
> () : unit
```