Learning to program with F#

Jon Sporring

September 7, 2016

Part V

Appendix

Appendix A

Number systems on the computer

A.1 Binary numbers

Humans like to use the *decimal number* system for representing numbers. Decimal numbers are *base* 10 means that for a number consisting of a sequence of digits separated by a *decimal point*, where each *digit* can have values $d \in \{0, 1, 2, \ldots, 9\}$ and the weight of each digit is proportional to its place in the sequence of digits w.r.t. the decimal point, i.e., the number $357.6 = 3 \cdot 10^2 + 5 \cdot 10^1 + 7 \cdot 10^0 + 6 \cdot 10^{-1}$ or in general:

· decimal number
· base
· decimal point
· digit

$$v = \sum_{i=-m}^{n} d_i 10^i \tag{A.1}$$

The basic unit of information in almost all computers is the binary digit or *bit* for short. A *binary* number consists of a sequence of binary digits separated by a decimal point, where each digit can have values $b \in \{0, 1\}$, and the base is 2. The general equation is,

· bit
· binary

$$v = \sum_{i=-m}^{n} b_i 2^i \tag{A.2}$$

and examples are $1011.1_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} = 11.5$. Notice that we use subscript 2 to denote a binary number, while no subscript is used for decimal numbers. The left-most bit is called the *most significant bit*, and the right-most bit is called the *least significant bit*. Due to typical organization of computer memory, 8 binary digits is called a *byte*, and 32 digits a *word*.
Other number systems are often used, e.g., *octal* numbers, which are base 8 numbers, where each digit is $o \in \{0, 1, \ldots, 7\}$. Octals are useful short-hand for binary, since 3 binary digits maps to the set of octal digits. Likewise, *hexadecimal* numbers are base 16 with digits $h \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\}$, such that $a_{16} = 10$, $b_{16} = 11$ and so on. Hexadecimals are convenient since 4 binary digits map directly to the set of octal digits. Thus $367 = 101101111_2 = 557_8 = 16f_{16}$. A list of the intergers 0–63 is various bases is given in Table A.1.

· most significant bit
· least significant bit
· byte
· word
· octal
· hexadecimal

A.2 IEEE 754 floating point standard

The set of real numbers also called *reals* includes all fractions and irrational numbers. It is infinite in size both in the sense that there is no largest nor smallest number and between any 2 given numbers there are infinitely many numbers. Reals are widely used for calculation, but since any computer only has finite memory, it is impossible to represent all possible reals. Hence, any computation performed on a computer with reals must rely on approximations. *IEEE 754 double precision floating-point format* (*binary64*), known as a *double*, is a standard for representing an approximation of reals using 64 bits. These bits are divided into 3 parts: sign, exponent and fraction,

· reals
· IEEE 754 double precision floating-point format
· binary64
· double

$$s\, e_1 e_2 \ldots e_{11}\, m_1 m_2 \ldots m_{52},$$

Dec	Bin	Oct	Hex	Dec	Bin	Oct	Hex
0	0	0	0	32	100000	40	20
1	1	1	1	33	100001	41	21
2	10	2	2	34	100010	42	22
3	11	3	3	35	100011	43	23
4	100	4	4	36	100100	44	24
5	101	5	5	37	100101	45	25
6	110	6	6	38	100110	46	26
7	111	7	7	39	100111	47	27
8	1000	10	8	40	101000	50	28
9	1001	11	9	41	101001	51	29
10	1010	12	a	42	101010	52	2a
11	1011	13	b	43	101011	53	2b
12	1100	14	c	44	101100	54	2c
13	1101	15	d	45	101101	55	2d
14	1110	16	e	46	101110	56	2e
15	1111	17	f	47	101111	57	2f
16	10000	20	10	48	110000	60	30
17	10001	21	11	49	110001	61	31
18	10010	22	12	50	110010	62	32
19	10011	23	13	51	110011	63	33
20	10100	24	14	52	110100	64	34
21	10101	25	15	53	110101	65	35
22	10110	26	16	54	110110	66	36
23	10111	27	17	55	110111	67	37
24	11000	30	18	56	111000	70	38
25	11001	31	19	57	111001	71	39
26	11010	32	1a	58	111010	72	3a
27	11011	33	1b	59	111011	73	3b
28	11100	34	1c	60	111100	74	3c
29	11101	35	1d	61	111101	75	3d
30	11110	36	1e	62	111110	76	3e
31	11111	37	1f	63	111111	77	3f

Table A.1: A list of the intergers 0–63 in decimal, binary, octal, and hexadecimal.

where s, e_i, and m_j are binary digits. The bits are converted to a number using the equation by first calculating the exponent e and the mantissa m,

$$e = \sum_{i=1}^{11} e_i 2^{11-i}, \tag{A.3}$$

$$m = \sum_{j=1}^{52} m_j 2^{-j}. \tag{A.4}$$

I.e., the exponent is an integer, where $0 \leq e < 2^{11}$, and the mantissa is a rational, where $0 \leq m < 1$. For most combinations of e and m the real number v is calculated as,

$$v = (-1)^s (1+m) 2^{e-1023} \tag{A.5}$$

with the exception that

	$m = 0$	$m \neq 0$
$e = 0$	$v = (-1)^s 0$ (signed zero)	$v = (-1)^s m 2^{1-1023}$ (subnormals)
$e = 2^{11} - 1$	$v = (-1)^s \infty$	$v = (-1)^s$ NaN (not a number)

· subnormals
· NaN
· not a number

where $e = 2^{11} - 1 = 11111111111_2 = 2047$. The largest and smallest number that is not infinity is thus

$$e = 2^{11} - 2 = 2046 \tag{A.6}$$

$$m = \sum_{j=1}^{52} 2^{-j} = 1 - 2^{-52} \simeq 1. \tag{A.7}$$

$$v_{\max} = \pm \left(2 - 2^{-52}\right) 2^{1023} \simeq \pm 2^{1024} \simeq \pm 10^{308} \tag{A.8}$$

The density of numbers varies in such a way that when $e - 1023 = 52$, then

$$v = (-1)^s \left(1 + \sum_{j=1}^{52} m_j 2^{-j}\right) 2^{52} \tag{A.9}$$

$$= \pm \left(2^{52} + \sum_{j=1}^{52} m_j 2^{-j} 2^{52}\right) \tag{A.10}$$

$$= \pm \left(2^{52} + \sum_{j=1}^{52} m_j 2^{52-j}\right) \tag{A.11}$$

$$\overset{k=52-j}{=} \pm \left(2^{52} + \sum_{k=51}^{0} m_{52-k} 2^{k}\right) \tag{A.12}$$

which are all integers in the range $2^{52} \leq |v| < 2^{53}$. When $e - 1023 = 53$, then the same calculation gives

$$v \overset{k=53-j}{=} \pm \left(2^{53} + \sum_{k=52}^{1} m_{53-k} 2^{k}\right) \tag{A.13}$$

which are every second integer in the range $2^{53} \leq |v| < 2^{54}$, and so on for larger e. When $e - 1023 = 51$, then the same calculation gives,

$$v \overset{k=51-j}{=} \pm \left(2^{51} + \sum_{k=50}^{-1} m_{51-k} 2^{k}\right) \tag{A.14}$$

which gives a distance between numbers of $1/2$ in the range $2^{51} \leq |v| < 2^{52}$, and so on for smaller e. Thus we may conclude that the distance between numbers in the interval $2^n \leq |v| < 2^{n+1}$ is 2^{n-52}, for $-1022 = 1 - 1023 \leq n < 2046 - 1023 = 1023$. For subnormals the distance between numbers are

$$v = (-1)^s \left(\sum_{j=1}^{52} m_j 2^{-j} \right) 2^{-1022} \tag{A.15}$$

$$= \pm \left(\sum_{j=1}^{52} m_j 2^{-j} 2^{-1022} \right) \tag{A.16}$$

$$= \pm \left(\sum_{j=1}^{52} m_j 2^{-j-1022} \right) \tag{A.17}$$

$$\overset{k=-j-1022}{=} \pm \left(\sum_{j=-1023}^{-1074} m_{-k-1022} 2^k \right) \tag{A.18}$$

which gives a distance between numbers of $2^{-1074} \simeq 10^{-323}$ in the range $0 < |v| < 2^{-1022} \simeq 10^{-308}$.

Appendix B

Commonly used character sets

Letters, digits, symbols and space are the core of how we store data, write programs, and comunicate with computers and each others. These symbols are in short called characters, and represents a mapping between numbers, also known as codes, and a pictorial representation of the character. E.g., the ASCII code for the letter 'A' is 65. These mappings are for short called character sets, and due to differences in natural languages and symbols used across the globe, many different character sets are in use. E.g., the English alphabet contains the letters 'a' to 'z', which is shared by many other European languages, but which have other symbols and accents for example, Danish has further the letters 'æ', 'ø', and 'å'. Many non-european languages have completely different symbols, where Chinese character set is probably the most extreme, where some definitions contains 106,230 different characters albeit only 2,600 are included in the official Chinese language test at highest level.
Presently, the most common character set used is Unicode Transformation Format (UTF), whose most popular encoding schemes are 8-bit (UTF-8) and 16-bit (UTF-16). Many other character sets exists, and many of the later builds on the American Standard Code for Information Interchange (ASCII). The ISO-8859 codes were an intermediate set of character sets that are still in use, but which is greatly inferior to UTF. Here we will briefly give an overview of ASCII, ISO-8859-1 (Latin1), and UTF.

B.1 ASCII

The *American Standard Code for Information Interchange* (*ASCII*) [4], is a 7 bit code tuned for the letters of the english language, numbers, punctuation symbols, control codes and space, see Tables B.1 and B.2. The first 32 codes are reserved for non-printable control characters to control printers and similar devices or to provide meta-information. The meaning of each control characters is not universally agreed upon.
The code order is known as *ASCIIbetical order*, and it is sometimes used to perform arithmetic on codes, e.g., an upper case letter with code c may be converted to lower case by adding 32 to its code. The ASCIIbetical order also has consequence for sorting, i.e., when sorting characters according to their ASCII code, then 'A' comes before 'a', which comes before the symbol '{'.

· American Standard Code for Information Interchange
· ASCII
· ASCIIbetical order

B.2 ISO/IEC 8859

The ISO/IEC 8859 report `http://www.iso.org/iso/catalogue_detail?csnumber=28245` defines 10 sets of codes specifying up to 191 codes and graphic characters using 8 bits. Set 1 also known as ISO/IEC 8859-1, Latin alphabet No. 1, or *Latin1* covers many European languages and is designed to be compatible with ASCII, such that code for the printable characters in ASCII are the same in ISO 8859-1. In Table B.3 is shown the characters above 7e. Codes 00-1f and 7f-9f are undefined in ISO 8859-1.

· Latin1

x0+0x	00	10	20	30	40	50	60	70
00	NUL	DLE	SP	0	@	P	‘	p
01	SOH	DC1	!	1	A	Q	a	q
02	STX	DC2	"	2	B	R	b	r
03	ETX	DC3	#	3	C	S	c	s
04	EOT	DC4	$	4	D	T	d	t
05	ENQ	NAK	%	5	E	U	e	u
06	ACK	SYN	&	6	F	V	f	v
07	BEL	ETB	’	7	G	W	g	w
08	BS	CAN	(	8	H	X	h	x
09	HT	EM	)	9	I	Y	i	y
0A	LF	SUB	*	:	J	Z	j	z
0B	VT	ESC	+	;	K	[	k	{
0C	FF	FS	,	<	L	\	l	|
0D	CR	GS	–	=	M	]	m	}
0E	SO	RS	.	>	N	ˆ	n	˜
0F	SI	US	/	?	O	_	o	DEL

Table B.1: ASCII

B.3 Unicode

Unicode is a character standard defined by the Unicode Consortium, `http://unicode.org` as the *Unicode Standard*. Unicode allows for 1,114,112 different codes. Each code is called a *code point*, which represents an abstract character. However, not all abstract characters requires a unit of several code points to be specified. Code points are divided into 17 planes each with $2^{16} = 65,536$ code points. Planes are further subdivided into named *blocks*. The first plane is called the *Basic Multilingual plane* and it are the first 128 code points is called the *Basic Latin block* and are identical to ASCII, see Table B.1, and code points 128-255 is called the *Latin-1 Supplement block*, and are identical to the upper range of ISO 8859-1, see Table B.3. Each code-point has a number of attributes such as the *unicode general category*. Presently more than 128,000 code points covering 135 modern and historic writing systems, and obtained at `http://www.unicode.org/Public/UNIDATA/UnicodeData.txt`, which includes the code point, name, and general category.

· Unicode Standard
· code point
· blocks
· Basic Multilingual plane
· Basic Latin block
· Latin-1 Supplement block
· unicode general category

A unicode code point is an abstraction from the encoding and the graphical representation of a character. A code point is written as“U+” followed by its hexadecimal number, and for the Basic Multilingual plane 4 digits are used, e.g., the code point with the unique name LATIN CAPITAL LETTER A has the unicode code point is “U+0041”, and is in this text it is visualized as ’A’. More digits are used for code points of the remaining planes.

The general category is used in grammars to specify valid characters, e.g., in naming identifiers in F#. Some categories and their letters in the first 256 code points are shown in Table B.5.

To store and retrieve code points, they must be encoded and decoded. A common encoding is *UTF-8*, which encodes code points as 1 to 4 bytes, and which is backward-compatible with ASCII and ISO 8859-1. Hence, in all 3 coding systems the character with code 65 represents the character ’A’. Another popular encoding scheme is *UTF-16*, which encodes characters as 2 or 4 bytes, but which is not backward-compatible with ASCII or ISO 8859-1. UTF-16 is used internally in many compiles, interpreters and operating systems.

· UTF-8
· UTF-16

Code	Description
NUL	Null
SOH	Start of heading
STX	Start of text
ETX	End of text
EOT	End of transmission
ENQ	Enquiry
ACK	Acknowledge
BEL	Bell
BS	Backspace
HT	Horizontal tabulation
LF	Line feed
VT	Vertical tabulation
FF	Form feed
CR	Carriage return
SO	Shift out
SI	Shift in
DLE	Data link escape
DC1	Device control one
DC2	Device control two
DC3	Device control three
DC4	Device control four
NAK	Negative acknowledge
SYN	Synchronous idle
ETB	End of transmission block
CAN	Cancel
EM	End of medium
SUB	Substitute
ESC	Escape
FS	File separator
GS	Group separator
RS	Record separator
US	Unit separator
SP	Space
DEL	Delete

Table B.2: ASCII symbols.

x0+0x	80	90	A0	B0	C0	D0	E0	F0
00			NBSP	°	À	Ð	à	ð
01			¡	±	Á	Ñ	á	ñ
02			¢	^{2}	Â	Ò	â	ò
03			£	^{3}	Ã	Ó	ã	ó
04			¤	´	Ä	Ô	ä	ô
05			¥	μ	Å	Õ	å	õ
06			¦	¶	Æ	Ö	æ	ö
07			§	·	Ç	×	ç	÷
08			¨	¸	È	Ø	è	ø
09			©	^{1}	É	Ù	é	ù
0a			ª	º	Ê	Ú	ê	ú
0b			«	»	Ë	Û	ë	û
0c			¬	$\frac{1}{4}$	Ì	Ü	ì	ü
0d			SHY	$\frac{1}{2}$	Í	Ý	í	ý
0e			®	$\frac{3}{4}$	Î	Þ	î	þ
0f			¯	¿	Ï	ß	ï	ÿ

Table B.3: ISO-8859-1 (latin1) non-ASCII part. Note that the codes 7f – 9f are undefined.

Code	Description
NBSP	Non-breakable space
SHY	Soft hypen

Table B.4: ISO-8859-1 special symbols.

General category	Code points	Name
Lu	U+0041–U+005A, U+00C0–U+00D6, U+00D8–U+00DE	Upper case letters
Ll	U+0061–U+007A, U+00B5, U+00DF–U+00F6, U+00F8–U+00FF	Lower case letter
Lt	None	Digraphic letter, with first part uppercase
Lm	None	Modifier letter
Lo	U+00AA, U+00BA	Gender ordinal indicator
Nl	None	Letterlike numeric character
Pc	U+005F	Low line
Mn	None	Nonspacing combining mark
Mc	None	Spacing combining mark
Cf	U+00AD	Soft Hyphen

Table B.5: Some general categories for the first 256 code points.

Appendix C

A brief introduction to Extended Backus-Naur Form

Extended Backus-Naur Form (*EBNF*) is a language to specify programming languages in. The name is a tribute to John Backus who used it to describe the syntax of ALGOL58 and Peter Nauer for his work on ALGOL 60.

· Extended Backus-Naur Form

An EBNF consists of *terminal symbols* and *production rules*. Examples of typical terminal symbol are characters, numbers, punctuation marks, and whitespaces, e.g.,

· EBNF
· terminal symbols

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
```

A production rule specifies a method of combining other production rules and terminal symbols, e.g.,

· production rules

```
number = digit { digit };
```

A proposed standard for ebnf (proposal ISO/IEC 14977, `http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf`) is,

'=' definition, e.g.,

```
zero = "0";
```

here `zero` is the terminal symbol `0`.

',' concatenation, e.g.,

```
one = "1";
eleven = one, one;
```

here `eleven` is the terminal symbol `11`.

';' termination of line

'|' alternative options, e.g.,

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
```

here `digit` is the single character terminal symbol, such as `3`.

'[...]' optional, e.g.,

```
zero = "0";
nonZeroDigit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
nonZero = [ zero ], nonZeroDigit;
```

here `nonZero` is a non-zero digit possibly preceded by zero, such as `02`.

'{ ... }' repetition zero or more times, e.g.,

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
number = digit, { digit };
```

here `number` is a word consisting of 1 or more digits, such as `12`.

'(...)' grouping, e.g.,

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
number = digit, { digit };
expression = number, { "+" | "-", number };
```

here `expression` is a number or a sum of numbers such as `3 + 5`.

'`" ... "`' a terminal string, e.g.,

```
string = "abc"';
```

"`' ... '`" a terminal string, e.g.,

```
string = 'abc';
```

'`(* ... *)`' a comment (* ... *)

```
(* a binary digit *) digit = "0" | "1"; (* from this all numbers may be
    constructed *)
```

Everything inside the comments are not part of the formal definition.

'`? ... ?`' special sequence, a notation reserved for future extensions of EBNF.

```
codepoint = ?Any unicode codepoint?;
```

'–' exception, e.g.,

```
letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H"
  | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q"
  | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z";
vowel = "A" | "E" | "I" | "O" | "U";
consonant = letter - vowel;
```

here `consonant` are all letters except vowels.

Rules for rewriting EBNF are:

Rule	Description
`s | t` ↔ `t | s`	`|` is commutative
`r | (s | t)` ↔ `(r | s)| t` ↔ `r | s | t`	`|` is associative
`(r s)t` ↔ `r (s t)` ↔ `r s t`	concatenation is associative
`r (s | t)` ↔ `r t| r s`	concatenation is distributive over `|`
`(r | s)t` ↔ `r t| r t`	
`[s | t]` ↔ `[t] | [s]`	
`[[s]]` ↔ `[s]`	`[]` is idempotent
`{{s}}` ↔ `{s}`	`{}` is idempotent

where `r`, `s`, and `t` are production rules or terminals. Precedence for the EBNF symbols are,

Symbol	Description
`*`	repetition
`-`	except
`,`	concatenate
`|`	option
`=`	define
`;`	terminator

in order of precedence, such that `*` has higher precedence than `-`. These precedence rules are overridden by bracket pairs, such as `' '`, `" "`, `(* *)`, `( )`, `[ ]`, `{ }`, `? ?`.
The proposal allows for identifies that includes space, but often a reduced form is used, where identifiers are single words, in which case the concatenation symbol `,` is replaced by a space. Likewise, the termination symbol `;` is often replaced with the new-line character, and if long lines must be broken, then indentation is used to signify continuation. In this relaxed EBNF, the EBNF syntax itself can be expressed in EBNF as,

```
letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H"
  | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q"
  | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
  | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h"
  | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q"
  | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z";
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
symbol = "[" | "]" | "{" | "}" | "(" | ")" | "<" | ">"
  | "?" | "'" | '"' | "=" | "|" | "." | "," | ";";
underscore = "_";
space = " ";
newline = ?a newline character?;
identifier = letter  { letter | digit | underscore };
character = letter | digit | symbol | underscore;
string = character  { character };
terminal = "'"  string  "'" | '"'  string  '"';
rhs = identifier
  | terminal
  | "["  rhs  "]"
  | "{"  rhs  "}"
  | "("  rhs  ")"
  | "?"  string  "?"
  | rhs  "|"  rhs
  | rhs  ","  rhs
  | rhs  space  rhs; (*relaxed ebnf*)
rule = identifier  "="  rhs  ";"
  | identifier  "="  rhs newline; (*relaxed ebnf*)
grammar = rule { rule };
```

Here the comments demonstrate, the relaxed modification. Newline does not have an explicit representation in EBNF, which is why we use `? ?` brackets