국어 정보화의 방향

문자 코드를 중심으로

박진호
서울대학교 국어국문학과 부교수

1. 국어 정보화의 기본 개념

이 글에서는 국어 정보화를 위해 어떤 노력을 기울여 왔는지, 앞으로 국어 정보화를 원활히 추진하기 위해 노력해야 할 점은 무엇인지 살펴보고자 한다. 이를 위해 먼저 국어 정보화란 무엇인지, 국어 정보화의 기본 개념들을 알아본다.

1.1. 무엇을 정보화한다는 것의 의미

컴퓨터는 모든 데이터와 명령을 2진수로 표상한다. 그것은 컴퓨터의 저장 장치와 연산 장치가 반도체로 이루어져 있고, 반도체를 구성하고 있는 가장 작은 단위의 기억 소자는 두 가지 상태(켜져 있는 상태와 꺼져 있는 상태. 즉 1과 0)에만 있을 수 있기 때문이다. 즉, 컴퓨터에서 행해지는 연산은 궁극적으로 '어떤 위치의 기억 소자 상태를 0에서 1로, 또는 1에서 0으로 바꾸라'는 원초적인 연산으로 구성되어 있다.

'X 정보화'란 X를 컴퓨터에서 구현할 수 있는 방안을 모색하고, X를 컴퓨터 및 네트워크에서 더 효율적으로 처리하고 유통시킬 수 있는 방

안을 모색하는 것이라고 할 수 있다. X를 컴퓨터에서 구현하는 데에는 입력, 내부 처리, 출력의 세 단계를 생각할 수 있다. 이것은 X가 무엇이든 다 적용될 수 있는 일반적인 개념이다.

X가 음악 또는 소리이면 입력은 공기 중에서 음파의 형태로 구현되는 소리 정보(아날로그 사운드)를 마이크 등의 입력 장치를 통해 흡수하는 것이 된다. 내부 처리는 아날로그 사운드에 포함된 여러 정보(파형, 주파수, 세기 등)를 일정한 규약에 따라 디지털화(2진수로 표현)하여 디지털 파일의 형태(wave, mp3 등)로 저장하고 유통하는 것을 말한다. 출력은 디지털 사운드 파일을 스피커, 이어폰 등의 출력 장치를 통해 아날로그 사운드(음파)로 재현하는 것이다.

X가 시각적 이미지이면 입력 단계에서는 이미지가 갖고 있는 시각 정보(아날로그 이미지)를 스캐너, 카메라 등의 입력 장치를 통해 흡수한다. 내부 처리는 아날로그 이미지에 포함된 여러 정보를 일정한 규약에 따라 디지털화하는 방법이 관건이 된다. 예컨대 아날로그 이미지가 펼쳐져 있는 공간을 일정한 수의 격자(화소)로 분해하고 각 격자의 색채를 디지털 형태로 표현할 수 있다. 한 가지 방법은 각 화소의 색채를 R · G · B의 세 차원으로 나누고, 각 차원을 0～255 사이의 정수로 표현하는 것이다. 이들 정수도 궁극적으로는 2진수로 표현되겠지만 말이다. 이런 방법이 여러 가지가 있어서 이미지 파일도 bmp, jpg, gif 등 여러 가지 포맷이 생겨났다. 이미지를 이런 파일 형태로 표상하면 매우 손쉽게 저장하고 유통할 수 있다. 출력 단계에서는 디지털 이미지 파일을 모니터, 프린터 등의 출력 장치를 통해 아날로그 이미지로 재현하게 된다.

1.2. 국어 정보화의 의미

X가 국어를 포함한 언어인 경우를 생각해 보면, 언어는 음성과 문자의 두 가지 매체로 표현된다. 음성 언어 정보를 매체의 특성까지 살려서 정보화하려면, 위에서 살펴본 음악 · 소리를 정보화하는 방법을 따르면 된다. 문자는 위에서 말한 것과는 다른 독특한 방법으로 정보화를 하게 된다. 이 글에서는 바로 여기에 초점을 맞추어 살펴본다. 즉, 국어 정보화의 두 가지 측면 중 문자 정보화에 초점을 맞추어 살펴본다.

문자 정보화의 입력 단계에서는 키보드나 터치 패드상의 가상 키보드 등의 입력 장치를 통해, (일정한 규약에 따라 정해진) 키 스트로크 연쇄와 문자 간의 매핑 관계에 따라 문자 정보를 입력한다. 내부 처리 단계에서는 미리 정해진 규약에 따라 문자를 2진수 형태로 변환하여 기억 장치에 저장하고, 텍스트 파일의 형태로 저장하고 유통한다. 출력 단계에서는 2진수 형태로 저장된 정보를 정해진 규약에 따라 문자로 매핑하여 각 문자를 모니터 · 프린터 등의 출력 장치에 시각적 형태의 문자로 재현한다. 하나의 문자가 폰트에 따라 다양한 시각적 형태로 드러날 수 있다.

위의 설명에서도 드러나듯이, 언뜻 생각하면 입력→내부 처리→출력의 순서로 일이 진행될 것 같지만, 실제로는 내부 처리를 어떻게 할 것인지가 미리 결정되어 있어야 입력을 포함한 모든 일이 제대로 이루어질 수 있다. 이 글에서는 국어 정보를 컴퓨터 내부에서 처리하는 방법에 초점을 맞추어 살펴보겠다.

2. 문자 코드의 기초: 아스키 코드

문자를 컴퓨터로 구현하기 위해서는 각 문자를 2진수로 어떻게 표상할지 결정되어야 한다. 각 문자와 2진수 사이의 매핑 관계를 정해 놓은 것을 문자 코드라고 한다.

0 또는 1의 한 자리 2진수 정보를 저장할 수 있는 단위를 비트(bit)라고 한다. 하드웨어적으로는 최소의 기억 소자가 이에 해당한다. 1비트의 저장 공간으로는 두 가지 기호밖에 구분할 수 없다. 8비트의 묶음을 1바이트(byte)라고 한다. 1바이트로 나타낼 수 있는 기호의 개수는 2^8=256개이다.

컴퓨터 기술이 가장 먼저 발달한 미국에서 문자 정보화를 추진할 때 역시 문자 코드 문제가 가장 기본적인 문제로 대두되었다. 미국에서 일상적으로 사용하는 문자는 로마자(영문자) 대문자와 소문자, 숫자(0～9), 기호 등을 모두 합쳐도 100개 남짓이어서, 모든 문자를 1바이트로 나타낼 수 있다. 미국국가표준기구(ANSI, American National Standard Institute)에서 이 문자들과 2진수 사이의 매핑 관계를 정해 놓았는데, 이것을 아스키(ASCII, American Standard Code for Information Interchange)라고 부른다.

아스키 코드에 영문자 대문자는 65～90, 소문자는 97～122에 배정되어 있고, 숫자는 48(0)～57(9)에 배당되어 있다. 여기서 A의 코드 값이 65라는 말의 의미를 좀 더 깊이 생각해 보자. 10진수 65를 2진수로 나타내면 1000001이 된다. 즉, 1바이트의 저장 공간에 01000001의 형태로 구현되는 것이다. 하드웨어에서 실제로 일어나는 일을 충실하게 묘사하려면 2진수가 가장 좋지만, 인간이 보기에는 불편하므로, 2^4=16진수로 나타내는 것이 일반적이다. 16진수를 표기할 때에는 앞에 '0x'를 붙

여서 10진수와 구별한다. 즉, A라는 문자의 코드 값은 0x41이다.

우리가 컴퓨터에서 메모장 같은 텍스트 에디터를 열고 키보드를 이용하여 'Hello!'라고 입력하면 메모리에는 '01001000 01100101 01101100 01101100 01101111 00100001'과 같이 2진수들의 연쇄로 기록되고, 저장 버튼을 누르면 이 2진수들의 연쇄가 하드디스크에 저장되는 것이다. 하드디스크에 저장해 둔 텍스트 파일을 불러와서 열면 각 2진수들이 미리 약속된 매핑 관계에 따라 모니터에 문자로 변환되어 표시된다.

3. 1바이트 문자와 2바이트 문자

한글은 현재 통용되고 있는 것만 따져도 수천 개나 되기 때문에, 1바이트 가지고는 모든 한글을 구별하여 나타낼 수 없다. 반면에 2바이트로는 모든 한글 문자들을 나타낼 수 있다. 2바이트는 16비트이고, 2^{16}=65,536개의 기호를 구분할 수 있으므로, 모든 한글을 다 표상하고도 남음이 있다. 예컨대 한글 문자 '가'는 '10110000 10100001(16진수로는 0xb0a1)'로 나타낼 수 있다.

그런데 컴퓨터가 연산을 수행할 때, 저장 공간에 기록되어 있는 문자가 1바이트짜리인지 2바이트짜리인지 어떻게 알까? 그것은 각 바이트의 첫 번째 비트(MSB, Most Significant Bit)를 보고 알 수 있다. 1바이트 문자는 MSB가 0이고, 2바이트 문자는 MSB가 1이라고 규약을 정해 놓으면 된다.

4. 문자 코드의 국가 표준

문자 코드의 표준이 없고 회사마다 기계마다 다르다면 A 회사에서 만든 컴퓨터/소프트웨어로 작성한 텍스트 문서를 B 회사에서 만든 컴퓨터/소프트웨어로 열어 보면 완전히 다른 문자로 보일 수도 있을 것이다. 즉, 호환성이 없게 된다. 정보의 원활한 유통과 처리를 위해서는 문자 코드의 표준이 필요하다. 그래서 미국에서는 아스키 코드를 문자 코드의 국가 표준으로 정했고, 그 후 전 세계에서 아스키 코드가 국제 표준처럼 자리 잡았다. 미국 이외의 국가에서도 일상생활에서 로마자를 많이 사용하므로 어차피 이들에 대한 문자 코드를 배정해 놓아야 하고, 문자 코드를 미국 국가 표준과 굳이 다르게 정해서 호환성을 떨어뜨릴 이유가 없기 때문이다.

아스키 문자 이외의 문자를 사용하는 국가, 특히 1바이트 문자뿐 아니라 2바이트 문자까지 사용해야 하는 나라에서는 문자 코드의 표준을 정하는 일이 그리 단순치 않을 수 있다. 경제성(저장 공간을 적게 차지할수록 좋음)과 효율성(연산을 빨리 수행할수록 좋음)이라는 두 가지 가치가 종종 충돌하기 때문이다.

5. 한글 문자 코드의 초기 역사

한글 문자 코드를 만들기 위한 노력이 시작된 초기에는 한글 한 글자를 3바이트로 표현하는 방안이 제안되었다. 즉, 한글 한 글자가 초성 1바이트, 중성 1바이트, 종성 1바이트로 이루어지는 것이다. 한글 한 글자가 초성, 중성, 종성의 세 부분으로 이루어져 있다는 사실을 반영한,

그 나름대로 합리적인 방안이기는 하지만 초창기에는 저장 공간이 비쌌기 때문에 저장 공간을 가능한 한 적게 차지하는 방안이 선호되었고, 2바이트로도 표현할 수 있는 것을 3바이트로 표현하는 방안은 환영받지 못했다.

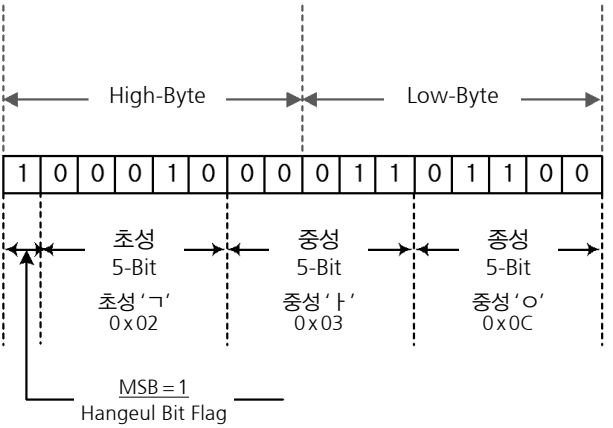
그 외에 가변 바이트 방안도 제안되었다. 이는 한글 한 글자를 구성하는 자소의 개수만큼 바이트를 배당하는 방법이다. 예컨대 '가'는 2바이트, '강'은 3바이트, '읽'은 4바이트가 된다. 그 나름대로 합리적인 방안이나 역시 경제성의 측면에서 환영받지 못했다. 그 후 한글의 구성 원리를 반영하면서도 보다 경제적인 방안이 모색된다.

6. 조합형 한글 코드

조합형 한글 코드(KSSM)는 한글 한 글자를 2바이트, 즉 16비트로 표현한다. 다만 첫 비트(MSB)는 1로 고정되어 있다. 2바이트 문자라는 사실을 알려 주어야 하기 때문이다. 나머지 15비트 중 앞 5비트는 초성, 그다음 5비트는 중성, 마지막 5비트는 종성을 나타낸다.

이 방안이 성공하려면 초성, 중성, 종성 모두 5비트로 나타낼 수 있어야 하는데, 다행히 5비트로 나타낼 수 있는 기호의 개수는 2^5=32가지이고, 한글의 초성은 19개, 중성은 21개, 종성은 28개이므로 모두 5비트로 표현 가능하다.

조합형은 2바이트라는 적은 저장 공간을 차지하면서도 초성, 중성, 종성으로 이루어져 있다는 한글의 구성적 특성을 잘 반영한다. 즉, 글자의 코드 값을 보고 약간의 연산을 수행하면 초성, 중성, 종성이 무엇인지 알 수 있다. 하나의 한글 글자를 분해하여 초성, 중성, 종성이 무엇



[그림 1] 조합형에서 '강'을 표상하는 방법

인지 알면, 보다 사용자 친화적인 인터페이스를 만들 수 있다. 예컨대 사용자가 어떤 단어를 입력하면 그 단어의 뜻을 알려 주는 일종의 사전 소프트웨어를 만든다고 할 때, 사용자가 입력한 단어가 사전에 들어 있지 않으면 "~은/는 사전에 안 들어 있습니다."라는 메시지를 출력해야 할 것이다. 이때 조사로 '은'을 쓸지 '는'을 쓸지는 해당 단어의 마지막 음절에 종성이 있는지 없는지를 알아야 결정할 수 있다.

[그림 1]에서 보는 바와 같이 '강'이라는 한글 글자는 초성 'ㄱ', 중성 'ㅏ', 종성 'ㅇ'으로 이루어져 있는데, 초성 'ㄱ'은 2진수 '00010(16진수 0x02)'으로 나타내고, 중성 'ㅏ'는 2진수 '00011(16진수 0x03)'로 나타내고, 종성 'ㅇ'은 2진수 '01100(16진수 0x0c)'으로 나타낸다. 즉, '강'의 코드 값은 2진수 '10001000 01101100(16진수 0x886c)'이 된다.

조합형 한글 코드에서 각 초성, 중성, 종성에 배당된 코드 값은 다음 표와 같다.

[표 1] 조합형에서 각 초성, 중성, 종성에 배당된 코드 값

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
초성	채움	ㄱ	ㄲ	ㄴ	ㄷ	ㄸ	ㄹ	ㅁ	ㅂ	ㅃ	ㅅ	ㅆ	ㅇ	ㅈ	ㅉ
중성		채움	ㅏ	ㅐ	ㅑ	ㅒ	ㅓ			ㅖ	ㅕ	ㅖ	ㅗ	ㅘ	ㅙ
종성	채움	ㄱ	ㄲ	ㄳ	ㄴ	ㄵ	ㄶ	ㄷ	ㄹ	ㄺ	ㄻ	ㄼ	ㄽ	ㄾ	ㄿ
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	
초성	ㅊ	ㅋ	ㅌ	ㅍ	ㅎ										
중성			ㅚ	ㅛ	ㅜ	ㅝ	ㅞ	ㅟ			ㅠ	ㅡ	ㅢ	ㅣ	
종성	ㅀ	ㅁ		ㅂ	ㅄ	ㅅ	ㅆ	ㅇ	ㅈ	ㅊ	ㅋ	ㅌ	ㅍ	ㅎ	

7. 완성형 한글 코드

조합형이 여러 장점이 있음에도 정부에서 국가 표준으로 채택한 것은 완성형 한글 코드였다(KSC 5601, KSX 1001). 조합형이 초성 · 중성 · 종성 코드를 조합 · 결합하는 방식인 데 비해 완성형은 그렇지 않다.(그래서 '완성형'이라는 명칭이 붙었다.)

완성형 역시 한글 한 글자를 2바이트로 표현한다. 그런데 이론적으로 가능한 1만 1,172자(초성 19×중성 21×종성 28) 전부에 코드를 배당하지 않고, 현대 일상생활에서 사용하는 글자 2,350자만을 추려서 코드를 배당하였다. 두 바이트 중 앞 바이트는 0xb0～0xc8(25개), 뒤 바이트는 0xa1～0xfe(94개)를 한글 영역으로 사용한다(25×94=2,350자). 한편 앞 바이트 0xca～0xfd(52개), 뒤 바이트 0xa1～0xfe(94개)의 영역은 한자를 위해 배당되었다. 즉, 완성형 한글 코드에서 한자는 52×94=4,888자가 표현 가능하다.

8. 완성형의 문제점

애초에 사용할 일이 없을 것으로 생각되어 2,350자에서 배제되었던 글자 중 일부를 사용할 필요가 생기게 되었다. 예컨대 〈똠방각하〉라는 소설이 나오고 드라마로도 만들어졌는데 '똠'은 2,350자에 들어 있지 않았다. 이런 글자들이 여럿 생기자 완성형(정확히 말하면 KSC 5601)에 대한 불만이 터져 나왔다.

마이크로소프트사는 윈도98을 출시하면서 이 문제를 보완하여 마이크로소프트 통합형 한글(Microsoft Unified Hangul) 코드를 내놓았다. 이론적으로 가능한 1만 1,172자 중 배제되었던 한글 8,822자에 새로 코드를 배당하게 된 것이다. 그런데 새로 배당하는 글자의 코드 영역이 기존 글자의 코드 영역과 겹치면 안 된다. 이것은 하위 호환성(backward compatibility)이라는 컴퓨터 세계의 대원칙 때문이다. 컴퓨터 관련 규약을 업데이트할 때, 이미 정해져 있던 것을 고치지 않고 새로운 것을 추가하기만 해야지, 기존의 약속을 깨뜨리면 안 된다는 것이다. 그래야만 과거의 규약에 입각해서 만들어진 데이터, 소프트웨어, 하드웨어 등이 새 버전에서도 유효하기 때문이다. 그래서 새로 코드를 배당받은 8,822자는 기존 코드 체계의 빈자리를 찾아 여기저기에 흩어져서 자리를 잡게 되었다.

그러다 보니 문자 코드 값 배열 순서가 한글 자모 배열 순서와 일치하지 않는 문제가 발생하게 되었다.

[표 2]에서 보듯이, '갂'은 '각'보다 자모순으로 뒤에 와야 하지만 코드 값은 '갂'이 '각'보다 앞에 오게 된 것이다. 이는 소팅에 문제를 불러온다. 컴퓨터는 기본적으로 코드 값에 따라 소팅을 한다. 그런데 코드 값에 따라 소팅을 하면 그 결과는 우리가 보기에 이상한 결과가 되는 것이

[표 2] 완성형에서 코드 순과 자모순의 불일치

연번	KS C 5601 코드	통합형 코드	한글
1	0xB0A1	0xB0A1	가
2	0xB0A2	0xB0A2	각
3	-	0x8141	갂
4	-	0x8142	갃
5	0xB0A3	0xB0A3	간
6	-	0x8143	갅
7	-	0x8144	갆
8	0xB0A4	0xB0A4	갇
9	0xB0A5	0xB0A5	갈
10	0xB0A6	0xB0A6	갉
11	0xB0A7	0xB0A7	갊
12	-	0x8145	갋
13	-	0x8146	갌
14	-	0x8147	갍
15	-	0x8148	갎
16	-	0x8149	갏

다. 물론 이 문제는 극복할 수 있는 문제이기는 하다. 그러나 먼 미래를 내다보고 문자 코드 정책을 정했으면 굳이 겪지 않아도 되는 불편을 감수하게 된 것이다.

완성형의 또 다른 문제점은 자소 분해를 하려고 할 때 드러난다. 조합형은 코드 자체만 보면 초성 · 중성 · 종성을 알 수 있지만, 완성형은 그것이 불가능하다. 따라서 완성형 한글에서 자소 분해를 하려면, 미리 완성형-조합형 매핑 테이블([그림 2] 참조)을 만들어 놓고, 분해하고자 하는 한글의 완성형 코드 값과 대응되는 조합형 코드 값을 얻어서 이를 바탕으로 자소 분해를 해야 한다.

1	0xb0a1,	0x8861,	가
2	0xb0a2,	0x8862,	각
3	0xb0a3,	0x8865,	간
4	0xb0a4,	0x8868,	갇
5	0xb0a5,	0x8869,	갈
6	0xb0a6,	0x886a,	갉
7	0xb0a7,	0x886b,	갊
8	0xb0a8,	0x8871,	감
9	0xb0a9,	0x8873,	갑
10	0xb0aa,	0x8874,	값
11	0xb0ab,	0x8875,	갓
12	0xb0ac,	0x8876,	갔
13	0xb0ad,	0x8877,	강
14	0xb0ae,	0x8878,	갖
15	0xb0af,	0x8879,	갗
16	0xb0b0,	0x887b,	같
17	0xb0b1,	0x887c,	갚
18	0xb0b2,	0x887d,	갛
19	0xb0b3,	0x8881,	개
20	0xb0b4,	0x8882,	객
21	0xb0b5,	0x8885,	갠
22	0xb0b6,	0x8889,	갤
23	0xb0b7,	0x8891,	갬
24	0xb0b8,	0x8893,	갭
25	0xb0b9,	0x8895,	갯
26	0xb0ba,	0x8896,	갰
27	0xb0bb,	0x8897,	갱
28	0xb0bc,	0x88a1,	갸
29	0xb0bd,	0x88a2,	갹
30	0xb0be,	0x88a5,	갼

[그림 2] 완성형-조합형 매핑 테이블

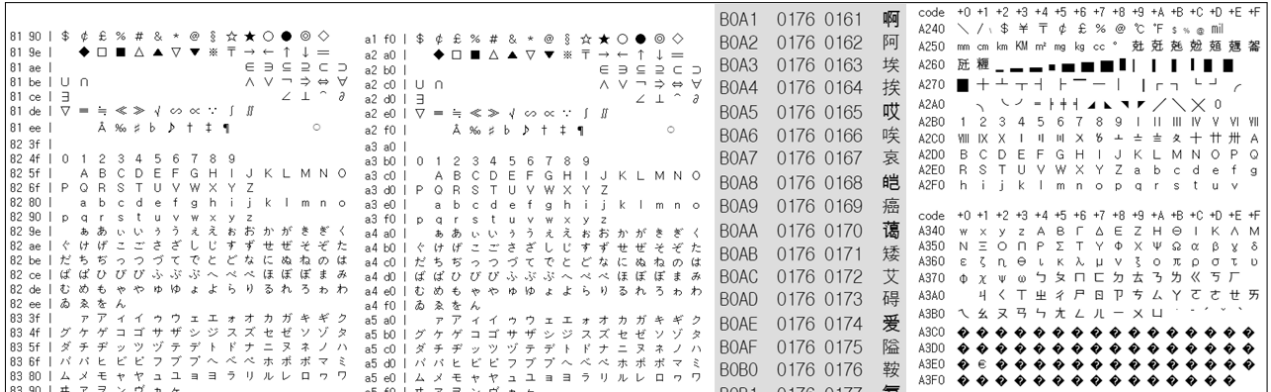
9. 일본, 중국, 대만의 문자 코드

한자를 많이 쓰는 일본, 중국, 대만에서도 한국과 비슷하게 1바이트 문자만으로는 부족하여, 나름의 2바이트 문자 코드 체계를 고안하여 사용하고 있다. 일본에서는 JIS(Japan Industrial Standard)라는 코드 체계와 이를 간소화한 시프트 JIS(Shift-JIS)를 많이 사용하고 있으며, 이 밖에 EUC(Extended Unix Characterset)-JP도 많이 사용한다. 간체(簡體)를 주로 사용하는 중국에서 채택된 표준은 GB라고 하고, 번체(繁體)를 주로 사용하는 대만에서 채택된 표준은 BIG-5라고 한다.

10. 문자 코드의 국가 간 충돌

각 나라는 다른 나라에서 어떤 글자에 어떤 코드 값을 배정했는지를 신경 쓰지 않고 (또는 신경을 썼더라도 어쩔 수 없이) 자기 나름대로 코드 값을 배정한다. 이에 따라 글자가 각 문자 코드 체계에서 서로 다른 코드 값을 배정받게 되었다. 거꾸로 말하면, 하나의 코드 값(code point)이 각 코드 체계에서 서로 다른 글자를 나타내게 되었다.

이에 따라 하나의 문서 내에서 한글, 한자(번체자, 간체자), 가나 등을 섞어서 쓰기가 어렵게 되었고, 모든 텍스트 문서는 "이 문서는 ~ 코드 체계에 따라 작성되었음"이라는 정보를 포함해야 국제적으로 유통될 수 있게 되었다. 이 정보가 누락되어 있는 경우, 소프트웨어가 인코딩을 오인하면 문자 깨짐 현상이 발생한다. 우리가 외국 인터넷 사이트를 방문할 때, 가끔 이런 문자 깨짐 현상을 경험하곤 한다. 그것은 해당 웹사이트의 HTML 파일의 헤더에 인코딩 정보가 누락되어 있기 때문이다.



[그림 3] 왼쪽부터 S-JIS, EUC-JP, GB, BIG-5 코드 체계 테이블

11. 유니코드의 대두

이러한 국가 간 코드 영역의 중복 문제를 해소하기 위해, 각국 대표들이 모여 세계의 모든 문자들을 동시에 표현할 수 있으면서 각 문자가 서로 겹치지 않는 통일된 문자 코드 체계를 만드는 방안을 논의하게 되었다. 이러한 움직임에는, 세계 공통의 문자 코드를 통해 자사 제품의 지역화(localization) 비용을 줄이려는 대규모 다국적 기업들의 강력한 후원도 한몫을 했다고 할 수 있다.

이렇게 해서 결성된 유니코드 컨소시엄에서 유니코드(Unicode)의 제정과 업데이트를 주관하고 있으며(http://www.unicode.org 참조), 국제 표준 기구(ISO)에서도 유니코드를 국제 표준으로 채택하였다(ISO/IEC 10646).

12. 유니코드의 문자 인코딩 방식

유니코드에서 문자를 인코딩하는 방식에는 UTF-32, UTF-16, UTF-8의 세 가지가 있다. UTF-32는 모든 문자를 4바이트=32비트로 표현한다. 따라서 2^{32}=4,294,967,296개(약 43억 개)의 문자들을 구별하여 표상할 수 있다. UTF-32는 모든 문자를 일관되게 4바이트로 표현하므로, 소프트웨어가 특별한 고려를 할 필요 없이 일관성 있게 문자를 처리할 수 있다는 장점이 있다. 그러나 하나의 문자를 저장하는 데 너무 많은 저장 공간이 소요되는 것이 단점이다. 반도체 가격이 빠른 속도로 싸지고 있으므로 저장 공간을 많이 차지하는 게 큰 문제가 아닐 것 같지만 대용량 서버를 운용해야 하는 기업 입장에서 저장 공간은 곧 돈과 직결된다.

UTF-16은 사용 빈도가 높은 글자는 2바이트로 표현하고 사용 빈도가 낮은 글자는 4바이트로 표현한다. 전자의 글자들은 BMP(Basic Multilingual Plane)에 속한다고 하고, 후자의 글자들은 SP(Supplementary Plane)에 속한다고 한다. 하나의 글자가 2바이트짜리인지 4바이트짜리인지를 컴퓨터가 구별할 수 있어야 하므로, BMP에 속하는 글자는 2^{16}=65,536개의 코드 값을 모두 사용할 수는 없다. 4바이트 문자가 사용하는 영역을 서로게이트(surrogate) 영역이라고 하는데, 앞의 2바이트(High-Surrogate)는 U+D800～U+DBFF(즉, 상위 6비트는 110110으로 고정되고, 나머지 10비트만 사용), 뒤의 2바이트(Low-Surrogate)는 U+DC00～U+DFFF(즉, 상위 6비트는 110111로 고정되고, 나머지 10비트만 사용)에 해당한다. BMP에 속하는 문자는 이 서로게이트 영역은 비워 두어야 한다. 그래도 영문자, 한글, 한자 등 상당수의 문자들을 BMP 내에서 표상할 수 있다.

사용 빈도가 높은 글자는 적은 저장 공간을 차지하고 사용 빈도가 낮은 글자는 많은 저장 공간을 차지하게 한다는 아이디어를 한걸음 더 발전시킨 것이 UTF-8이다. UTF-8에서는 하나의 글자가 1～4바이트로 표상된다. 하나의 글자가 몇 바이트로 표현되는가는 첫 바이트의 상위 비트(들)로 표현한다. 1바이트 문자의 MSB는 0, 2바이트 문자의 첫 바이트의 상위 비트는 110, 3바이트 문자의 첫 바이트의 상위 비트는 1110, 4바이트 문자의 첫 바이트의 상위 비트는 11110으로 고정되어 있다. 멀티 바이트(multi-byte) 문자의 첫 바이트가 아닌 나머지 바이트들은 상위 비트가 10이다. 이는 한 문자에 대한 바이트 표현이 다른 문자에 대한 바이트 표현의 일부가 되는 경우가 없도록 하기 위한 것이다. 만약 그런 경우를 허용한다면, 문자열(string) 내에서 하위 문자열(substring)을 찾는 알고리즘이 매우 복잡해진다. UTF-8의 장점은 아스키와 호환

[표 3] UTF-16과 UTF-8의 대응 관계

코드 범위(16진수)	UTF-16 (2진수)	UTF-8 (2진수)	설명
000000-00007F	00000000 0xxxxxxx	0xxxxxxx	ASCII와 동일한 범위
000080-0007FF	00000xxx xxxxxxxx	110xxxxx 10xxxxxx	첫 바이트는 110 또는 1110으로 시작하고, 나머지 바이트들은 10으로 시작함
000800-00FFFF	xxxxxxxx xxxxxxxx	1110xxxx 10xxxxxx 10xxxxxx	
010000-10FFFF	110110yy yyxxxxxx 110111xx xxxxxxxx	11110zzz 10zzxxxx 10xxxxxx 10xxxxxx	UTF-16 서로게이트 쌍 영역 (yyyy = zzzzz - 1). UTF-8로 표시된 비트 패턴은 실제 코드 포인트와 동일

가능(ASCII-compatible)하다는 것이다. 즉, 아스키 코드 체계에서 코드를 배정받은 문자들은 UTF-8에서도 동일한 코드 값을 가진다. 그 덕분에 아스키 코드에 입각해서 만들어진 소프트웨어로 UTF-8 텍스트를 처리할 수 있다. 반면에, 하나의 글자를 표현하는 데 소요되는 바이트 수가 들쭉날쭉하므로, 컴퓨터나 프로그래머가 이에 대해 신경을 써야 한다는 것이 단점이다.

13. 유니코드와 한글

이론적으로 가능한 현대 한글 1만 1,172자 전부가 BMP의 코드 영역 0xAC00～0xD7A3에 배당되어 있다. BMP에서 한글은 한자 다음으로 넓은 영역을 차지하고 있는 것이다.

유니코드의 한글은 초성 · 중성 · 종성 조합식은 아니나, 자모순대로 체계적으로 배열되어 있어서 계산에 의한 자모 분해가 가능하다. 한글 글자의 코드 값을 X라 하면

초성값=(X-0xAC00)/588

Y=(X-0xAC00)% 588

중성값=Y/28

종성값=Y % 28=(X-0xAC00)% 28

과 같은 간단한 식에 의해 초성 · 중성 · 종성을 알아낼 수 있다. KSC 5601의 전철을 밟지 않고 1만 1,172자 모두 코드 값을 배당하여, 한글의 컴퓨터 처리가 원활히 이루어지게 되었다. 자음과 모음 각 자소(옛한글 포함)도 따로 코드를 배정받았다.

14. 옛한글과 구결자의 정보화

국어사 자료에는 옛한글, 구결자 등 특수한 문자가 많이 사용되는데, 아스키 · 완성형 · 조합형 등의 코드 체계에서는 옛한글, 구결자를 표상할 수 없다. '보석글'이라는 워드프로세서에서 처음으로 옛한글에 코드를 배당하여 표상하기 시작했고, 한글과컴퓨터사의 '흔글' 워드프로세서의 HNC 코드에서도 옛한글에 코드를 배당하였다.

옛한글에 이어 구결자도 코드를 배정받게 되었다. 1992년 '흔글 2.0 전문가용'에서 HNC 코드번호 1D00～1DF3(244자)이 처음으로 배정되었고, '흔글 3.01'에서 16자가 추가되었으며(HNC 코드번호 1DF4～1E03),

‘ᄒᆞᆫ글 97’에서 9자가 추가되어(HNC 코드번호 1E04～1E0C), 총 269자를 표상할 수 있게 되었다.

15. 유니코드와 옛한글

2000년 무렵 한글과컴퓨터사(ᄒᆞᆫ글), 한국마이크로소프트(MS 워드), 삼성전자(훈민워드)는 각각 자사의 워드프로세서 제품을 유니코드 기반으로 뜯어고치는 과정에서, 유니코드에서 코드를 배정받지 못한 옛한글, 구결자 등을 처리하는 방식을 3사가 통일하기로 합의했다. 문자 코드가 통일되어 있어야, 타사 제품으로 작성한 문서를 자사 제품에서 읽을 수 있기 때문이다. 그래서 사용자 정의 영역(private use area)에 옛한글과 구결자를 배당하였다. 옛한글은 U+E0BC～U+F66E 영역에 5,299자가 배당되었고, 그 외의 옛한글은 초성 · 중성 · 종성 조합식 총 6바이트로 표상하게 되었다(옛한글 자소 영역: U+F785～U+F8F7). 구결자는 U+F67E～U+F77c 영역에 255자가 배당되었다. HNC의 269자 중 자형이 동일한 것은 통합하였다.

그리고 이를 화면상에 표현할 수 있도록 글꼴도 제작하였다. ‘ᄒᆞᆫ글’의 한컴돋움, 한컴바탕 등, 마이크로소프트 오피스의 새굴림, 새돋움, 새바탕, 새궁서 등이 그런 글꼴이다.

사용자 정의 영역을 이용한 것은 당시로서 어쩔 수 없는 선택이기는 했으나, 이는 어디까지나 임시방편이라고 할 수밖에 없다. 국제적 통용성이 전혀 없는 것이다.

16. 옛한글 표현 방식의 변화

사용자 정의 영역을 사용하는 것은 임시방편에 불과했기 때문에, 마이크로소프트 워드 2007, 흔글 2010부터는 옛한글 표현 방식이 조합식으로 바뀌게 되었다. 유니코드의 BMP에 코드 값을 배정받은 초성 · 중성 · 종성을 조합하여 하나의 한글 글자(음절)를 표현하는 것이다. 이에 따라 하나의 글자를 표현하는 데 소요되는 바이트 수가 일정하지 않게 되었다. 종성이 없는 옛한글 글자는 초성 2바이트+중성 2바이트=4바이트, 종성이 있는 옛한글 글자는 초성 2+중성 2+종성 2=6바이트, 현대 한글은 모두 2바이트로 표상된다. 하나의 글자가 차지하는 바이트 수가 일정하지 않을 뿐 아니라 옛한글 한 글자를 표현하는 데 소요되는 바이트 수가 현대 한글보다 많아 저장 공간을 많이 차지한다.

더 큰 문제는 하위 호환성(backward compatibility)이 깨졌다는 것이다. 구버전(마이크로소프트 워드 2003 이하, 흔글 2007 이하)에서 작성한 옛한글 문서와 신버전에서 작성한 옛한글 문서의 문자 코드가 달라 호환성에 문제가 발생한다. 흔글 2010에서 입력한 옛한글은 조합식이므로, 흔글 2007 이하 버전에서 열면 초성 · 중성 · 종성이 조합되지 않고 따로따로 나온다.

17. 북한의 문자 코드

북한의 문자 코드 표준은 KPS 9566(국규 9566)이라 불린다. 여기서 상위 바이트 0xA1～0xFE, 하위 바이트 0xA1～0xFE에 해당하는 2바이트 문자의 코드를 배당해 놓았다(94행×94열=8,836자). 16～44행에 한

[표 4] 북한의 자모순

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
초성	ㄱ	ㄴ	ㄷ	ㄹ	ㅁ	ㅂ	ㅅ	ㅈ	ㅈ	ㅊ	ㅋ	ㅌ	ㅍ	ㅎ
중성	ㅏ	ㅑ	ㅓ	ㅕ	ㅗ	ㅛ	ㅜ	ㅠ	ㅡ	ㅣ	ㅐ	ㅒ	ㅔ	ㅖ
종성	채움	ㄱ	ㄲ	ㄴ	ㄵ	ㄶ	ㄷ	ㄹ	ㄺ	ㄻ	ㄼ	ㄽ	ㄾ	ㄿ
	15	16	17	18	19	20	21	22	23	24	25	26	27	28
초성	ㄲ	ㄸ	ㅃ	ㅆ	ㅉ	ㅇ								
중성	ㅚ	ㅟ	ㅢ	ㅘ	ㅝ	ㅙ	ㅞ							
종성	ㅀ	ㅁ	ㅂ	ㅄ	ㅅ	ㅇ	ㅈ	ㅊ	ㅋ	ㅌ	ㅍ	ㅎ	ㄲ	ㅆ

행 \ 렬	72	73	74	75	76	77	78	79	80	81
1	⌒	⌒	⌒	⌒	⌒	⌒	⌐	⌐	⌐	⌐
2	▽	▼	▷	◁	▶	◀	○	●	❖	◭
3	h	i	j	k	l	m	n	o	p	q
4	김	일	성	김	정	일				
5	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
6	Ⅷ	Ⅸ	Ⅹ							i
7	⁸	⁹	½	⅓	⅔	¼	¾		₀	₁
8	Pa	kPa	MPa	GPa	ℓ	μℓ	mℓ	dℓ	kℓ	gal
9										

[그림 4] KPS 9566의 특수 문자 영역

[그림 5] 북한 운영 체제 '붉은별 3.0 서버용'의 특수 문자 영역

글 글자들이 북한의 자모순대로 배당되어 있고, 45～94행에 한자 4,653자가 배당되어 있다. 한자의 배열 순서도 역시 각 한자 독음의 북한 자모순이다.

KPS 9566-97에서는 한글 2,679자만 배당했으나, KPS 9566-2003에서는 나머지 8,493자(=11,172-2,679)를 사이사이 비어 있는 영역에 배당했다. 한국의 완성형 코드의 전철을 북한도 똑같이 밟은 것이다.

KPS 9566의 특징 중 하나는 한글 몇 글자가 특수 문자 영역에 특별히 배당되어 있다는 것이다.

북한에서 2013년 개발된 운영 체제 '붉은별 3.0 서버용'에서는, 이렇게 특별 대접을 받는 한글 글자에 3자가 추가되었다.

유니코드의 대두 이후, 북한에서도 국가 표준인 KPS 9566 이외에 유니코드도 함께 사용하고 있다. 그런데 유니코드에 들어 있는 한글 1만 1,172자는 한국이 제안한 것이므로, 북한의 자모순이 아니라 한국의 자모순을 따르고 있다. 북한에서는 이것이 마음에 안 들어서 그랬는지, 한때 자기들 나름대로 순서를 재배열해서 사용했었다. 그러나 2010년 나온 운영 체제 '붉은별'부터는 유니코드 한글을 표준대로 사용하고 있다. 자모 배열 순서나 문자 코드에 대해 남측과 북측이 일대일로 협상을 해서 타협안을 만들어 내는 것은 매우 어려운 일일 터이나, 유니코드처럼 국제적으로 통용되는 표준이 있으면 자연스럽게 이쪽으로 통일이 이루어질 수 있을 것이다.

18. 한글 문자 코드의 역사에서 얻을 수 있는 교훈

지금까지 살펴본 한글 관련 문자 코드의 역사는, 국어 정보화의 방향을 생각하는 데 많은 시사점을 제공한다. 첫째, 국어와 한글을 정보화할 때 한글의 특성을 컴퓨터에서도 잘 반영할 수 있게 해야 한다. 완성형보다는 조합형이 한글의 구조적 특성을 잘 반영한다. 둘째, 정보화 정책은 먼 미래를 내다볼 수 있어야 한다. KSC 5601에서 한글 2,350자만 코드값을 배당한 것은 근시안적 결정이었다. 셋째, 하위 호환성(backward compatibility)을 깨지 않는 것이 좋다. 이것이 깨지면 구버전에 입각한 데이터와 신버전에 입각한 데이터가 호환 불가능하게 된다. 옛한글 처리 방식의 변화로 호환성이 깨진 것이 대표적인 사례이다. 넷째, 국제 표준(예: 유니코드), 국제 동향에 민감 · 신속 · 적절히 대응해야 하며, 특별한 이유가 없다면 국제 표준을 따르는 것이 좋다. 옛한글과 구결자

를 사용자 정의 영역에 배당한 것은 국제 표준에 둔감한, 국제화에 역행하는 조처였다. 옛한글과 구결자는 이들 글자를 지원하는 폰트가 설치되어 있어야만 제대로 출력된다. 해당 폰트가 설치되어 있지 않은 컴퓨터(예컨대 외국)에서는 옛한글이 제대로 보이지 않는다. 한국의 전통문화를 세계화하고 외국인들에게 알리는 일을 원활히 하기 위해서는, 문자 코드라는 인프라가 국제 표준에 맞게 잘 되어 있어야 한다.