Internationalisierung mit Unicode

In fremden Zungen

Die Internationalisierung von Software wird immer wichtiger. Unicode verspricht Hilfe bei der Verwendung der passenden Schriftsysteme, egal ob Arabisch, Asiatisch oder Europäisch. Dieser Artikel gibt einen Überblick über die Schwierigkeiten und Unicodes Lösungsansätze. Bruno Haible, Oliver Frommel

Dass ein und dieselbe Software auf der ganzen Welt eingesetzt wird, hat Konsequenzen. Sie muss an die Gepflogenheiten verschiedener Länder anpassbar sein. Das reicht vom Übersetzen der Programmmeldungen und GUI-Elemente über das Datumsformat bis zur landesspezifischen Postleitzahl-Stadt-Land-Eingabemaske. Diese einzelnen Maßnahmen werden unter dem Stichwort Internationalisierung zusammengefasst, kurz I18N – die Zahl 18 steht dabei für die Anzahl der ausgelassenen Buchstaben zwischen I und n des englischen Worts Internationalization.

Eine Datei, viele Sprachen

Mittlerweile gibt es auch wachsenden Bedarf dafür, Texte verschiedener Sprachen gleichzeitig in ein und demselben Dokument zu verwenden: In dem jeweiligen Webbrowser sollen Suchmaschinentreffer in mehreren Sprachen auf einer einzigen Seite dargestellt werden. Datenbank-Benutzer wiederum wollen Postanschriften verschiedenster Länder abspeichern, auch wenn sie nur in der Landesschrift formulierbar sind. Technische Redakteure nutzen eine Textverarbeitung etwa für die mehrsprachige Bedienungsanleitung zu einer Mikrowelle. Von dieser Multilingualisierung (Mehrsprachigkeit) sind also unterschiedlichste Softwaregattungen betroffen.

Einheit statt Wildwuchs

Der Unicode-Standard [1] ist eine der Grundlagen für robuste Internationalisierung und gewissermaßen der Königsweg der Multilingualisierung. Natürlich gab es schon früher Ansätze dazu, aber meist nur mit eingeschränktem Anwendungsgebiet, oder eben mehrere, konkurrierende Lösungen.

Beispielsweise ist das Multiplexen von 8-Bit- und 16-Bit-Zeichensätzen, mit Umschaltung zwischen beiden Modi, am Verschwinden. Außer in ISO-2022-JP-2 (einem speziellen Encoding für japanische E-Mails) und Emacs wird diese Methode wegen ihrer Komplexität nicht mehr benutzt.

Unicode beschreibt zurzeit rund 250 000 Zeichen und hat eine theoretische Kapazität von maximal 1 112 047 Zeichen. Darin sind die europäischen Buchstabenschriften (Lateinisch, Griechisch und Kyrillisch) ebenso enthalten wie ostasiatische Silbenschriften (für Chinesisch, Japanisch und Koreanisch) und Alphabete für alle anderen lebenden Sprachen. Hinzu kommen Sonderzeichen aller Art: römische Ziffern, Schachfiguren, das phonetische Alphabet und das Braille-Blindenalphabet, mathematische Relationen und Grafiksymbole für Block und Linien.

Der Begriff des Zeichens in Unicode ist ziemlich weit: Einzelne Akzente wie ’^` sind genauso Zeichen wie ihre Kombinationen mit Einzelbuchstaben, zum Beispiel é ù ñ ä. Steuerzeichen gehören ebenso zu Unicode wie Abschnittstrennzeichen und ein paar Richtungswechselzeichen für von rechts nach links geschriebene Schriften (Hebräisch, Persisch und Arabisch).

Zeichen und Glyphs

Unicode repräsentiert nur die Zeichen (Characters) selbst und nicht ihre konkrete Darstellung (Glyphs), die ist Sache des verwendeten Fonts (siehe [2] und [3]. So besteht auf Unicode-Ebene zwischen **Wort** und Wort kein Unterschied; die Unterschiede zwischen Fraktur- und lateinischer Schrift liegen nur beim Font. Daher sind übrigens Chinesen und Japaner gezwungen, verschiedene Fonts zu verwenden: Ein paar Dutzend unter den Zigtausenden von Schriftzeichen werden in China und Japan unterschiedlich geschrieben.

In den europäischen und ostasiatischen Sprachen gibt es zu jedem Zeichen ein Glyph und umgekehrt. Fürs Arabische

und viele indischen Schriften gilt diese Regel nicht. Ein Zeichen hat je nach Kontext verschiedene Schriftbilder, zum Beispiel abhängig davon, ob es am Wortanfang, -mitte oder Ende auftritt.
Genauso kann ein einziges Schriftbild bei verschiedenen Zeichen auftauchen, die Programme natürlich unterscheiden müssen. Zum Beispiel sehen das kyrillische Ve »U+412 CYRILLIC CAPITAL LETTER VE« und das griechische Beta »U+0392 GREEK CAPITAL LETTER BETA« großgeschrieben aus wie das im Deutschen verwendete lateinische große B »U+0042 LATIN CAPITAL LETTER B«. Dennoch handelt es sich um unterschiedliche Zeichen, entsprechend unterscheiden sich ihre Kleinbuchstaben.

Ordnung nach Ebenen

Der Standard legt alle Zeichen einer Schrift fest und bestimmt, wie komplexe Zeichen sich aus einfachen zusammensetzen. Jedes ist über eine Zahl, den Unicode Scalar Value oder auch Codepoint, eindeutig zu identifizieren. Konventionell stellt man diesem Code ein »U+« voran. Jedes Zeichen besitzt eine für Menschen lesbare Kurzbeschreibung, eine Art Namen. Folgende Zeile

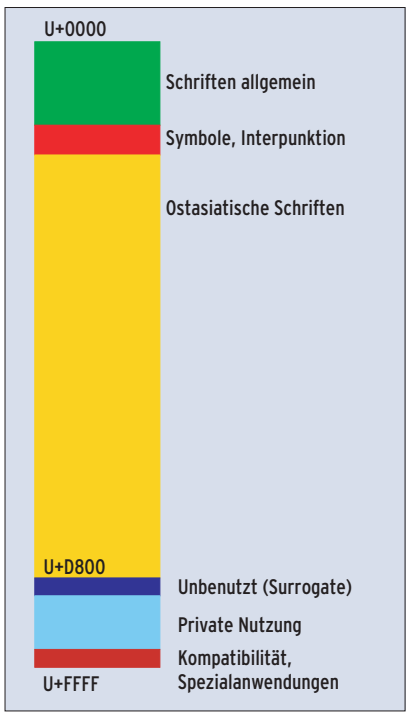

Abbildung 1: Die Basic Multilingual Plane enthält die wichtigsten Schriften und reicht vom Hexadezimalwert 0x0000 bis 0xFFFF.

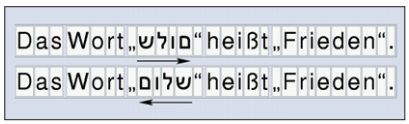

Abbildung 2: Gegenüber der logischen Reihenfolge in der Datei (oben) dreht der Renderer bei der Anzeige die Richtung des hebräischen Worts um.

repräsentiert das aus dem Deutschen bekannte kleine a:

```
U+0061 LATIN SMALL LETTER A
```

Um etwas Ordnung in die vielen Zeichen zu bringen, ist der riesige lineare Zahlenraum in 17 so genannte Planes (Ebenen) eingeteilt, von denen jede 64 K (2^{16}) Codepoints umfasst. Die erste heißt Basic Multilingual Plane (BMP) und enthält die am häufigsten verwendeten Schriften (Abbildung 1), die anderen Ebenen sind die Supplementary Planes. Die Codepoints der Surrogate sind für spezielle Anwendungen reserviert, um beispielsweise mit der 16-Bit-Kodierung UTF-16 auch Zeichen außerhalb der BMP anzusprechen.

Repräsentation als Bytes

Ein einzelnes Unicode-Zeichen als Codepoint ist also nur eine einfache Zahl. Wie werden solche Zeichen als Bytes repräsentiert, damit sie in Dateien, Webseiten und Datenbanken eingesetzt werden können? Die übliche Kodierung UTF-8 zerlegt jedes Unicode-Zeichen in Blöcke von drei bis sieben Bit und verpackt jeden dieser Blöcke in ein Byte,

Abbildung 3: Ein Unicode-Renderer muss mehr als einen Akzent pro Zeichen beherrschen.

siehe Tabelle 1. Diese Kodierung ist voll Ascii-kompatibel, denn die UTF-8-Bytes im Bereich 0x00 … 0x7F entsprechen den Ascii-Werten. Damit können C-Programme weiterhin die klassischen, mit »0x00« beendeten »char«-Strings verwenden. Ebenso gibt es bei der Übergabe von Dateinamen in UTF-8 an den Kernel kein Problem, weil »0x2F« für »/« steht und »0x2E« für ».«. Internetprotokolle machen wenig Schwierigkeiten, denn die meisten von ihnen können ohne weiteres Bytes im Bereich von 0x80 bis 0xFF transportieren.
Die UTF-8-Darstellung ist maschinenunabhängig, AMD- und Intel-Prozessoren behandeln die Bytes also in derselben Reihenfolge wie Sparc- und PowerPC-Prozessoren. Die Sortierung von UTF-8-Strings als Bytestrings ist identisch mit ihrer Sortierung nach Unicode-Codes; dies ist für die interne Konsistenz beispielsweise in Datenbanken wichtig. Dagegen ist die Reihenfolge, die man als Benutzer etwa bei »ls -l« sieht, von der Lokale abhängig und hat nichts mit der UTF-8-Kodierung zu tun.
Für die speicherinterne Darstellung kommen manchmal auch UTF-16, eine Serialisierung in 16-Bit-Blöcken, und UTF-32, eine Serialisierung in 32-Bit-Blöcken, zum Einsatz, siehe Tabellen 2 und 3. Beide werden aber selten benutzt, um

Tabelle 1: UTF-8

Wertebereich	Bit-Pattern	Byte-Darstellung
0x00..0x7F	$b_6 \dots b_0$	$\lvert 0b_6 \dots b_0 \rvert$
0x80..0x7FF	$b_{10} \dots b_0$	$\lvert 110b_{10} \dots b_6 \rvert 10b_5 \dots b_0 \rvert$
0x800..0xFFFF	$b_{15} \dots b_0$	$\lvert 1110b_{15} \dots b_{12} \rvert 10b_{11} \dots b_6 \rvert 10b_6 \dots b_0 \rvert$
0x10000..0x10FFFF	$b_{20} \dots b_0$	$\lvert 11110b_{20} \dots b_{18} \rvert 10b_{17} \dots b_{12} \rvert 10b_{11} \dots b_6 \rvert 10b_5 \dots b_0 \rvert$

Tabelle 2: UTF-16

Wertebereich	Bit-Pattern	16-Bit-Blöcke
0x0000..0xFFFF	$b_{15} \dots b_0$	$\lvert b_{15} \dots b_0 \rvert$
0x10000..0x10FFFF	0x10000 + [$b_{19} \dots b_0$]	$\lvert$ 0xD800 + [$b_{19} \dots b_{10}$] $\rvert$ 0xDC00 + [$b_9 \dots b_0$] $\rvert$

Tabelle 3: UTF-32

Wertebereich	Bit-Pattern	32-Bit-Blöcke
0x0000..0x10FFFF	$b_{20} \dots b_0$	$\lvert$ 00000000000 $b_{20} \dots b_0 \rvert$

Unicode-Daten zu speichern, weil die Darstellung maschinenabhängig ist: Was auf dem einen Computer »0x20 0xAC« ist, ist auf dem anderen »0xAC 0x20«.

Komplexes Rendering

Unicode-Text anzeigen erfordert einen leistungsfähigen Renderer, denn jede Schrift stellt besondere Anforderungen. Der Renderer ist die Bibliothek, die ein Stück Text mit Hilfe von Fonts darstellt. Der Standard muss viele Forderungen erfüllen: für die ostasiatischen Schriften 24 mal 24 Pixel große Fonts unterstützen; Schriften von rechts nach links für Hebräisch, Persisch und Arabisch; mehrere Akzente pro Buchstabe fürs Vietnamesische; Umordnung von Vokalen und Konsonanten für die indischen Sprachen und so weiter. Wegen der speziellen Regeln für die Rechts-nach-links-Schriften muss ein Unicode-Renderer immer absatzweise rendern. Es ist nicht möglich, einen Absatz Zeile für Zeile unabhängig voneinander zu rendern

In einer Datei stehen die Codes in logischer Reihenfolge, der Renderer ordnet sie für Rechts-links-Schriften bei der Anzeige um, siehe Abbildung 2. Auch mit mehreren Akzenten pro Zeichen muss der Renderer klarkommen, egal ob sie darüber oder darunter angeordnet sind, siehe Abbildung 3. In manchen Sprachen gibt es Vokale, die im Geschriebenen links und rechts an andere Zeichen angeschlossen werden.

Klar also, dass ein Unicode-Renderer viel Wissen über Zeichen, Glyphs und die Sprache selbst braucht. Hinweise, auch zu den Eigenheiten jeder Sprache, liefern die Dokumente des Unicode-Konsortiums [1]. Gute Renderer finden sich heute in den GUI-Toolkits QT und GTK/Pango, die beide auf der Xft-Bibliothek basieren.

Schwieriger Textvergleich

Mit dem Standard für einzelne Zeichen ist es nicht getan. Wichtig sind standardisierte Prozeduren, um Unicode-Strings zu ordnen und zu vergleichen. Unter dem Begriff Collation (Textvergleich) legt Unicode fest, wie eine Software die Sortierordnung von Zeichenketten bestimmt. Die Unicode-Dokumente beschreiben nicht nur das Ergebnis solcher Operationen, sondern auch passende Algorithmen.

Die so genannten Normalisierungsformen sorgen dafür, dass gleichbedeutende Strings von Programmen auch als gleich erkannt werden [4]. Ein zusammengesetztes Zeichen wie »U+0041 LATIN CAPITAL LETTER A, U+0301 COMBINING ACUTE ACCENT« soll in normalen Anwendungen gleichbedeutend zu dem entsprechenden »U+00C1 LATIN CAPITAL LETTER A WITH ACUTE« sein. Glücklicherweise gibt es auch dafür fertige Bibliotheken, die dem Programmierer Arbeit abnehmen, etwa die International Components for Unicode (ICU) von IBM [5], die es für C/C++ und Java gibt.

Problem: Eintippen

Zur Eingabe von Unicode-Text dienen Eingabemethoden (Input Methods, IM). Im einfachsten Fall kann man damit auf einer amerikanischen Tastatur deutsche Umlaute eingeben: Der Benutzer tippt »a"« und auf dem Bildschirm erscheint »ä«. Komplizierter ist es bei CJK-Sprachen (Chinesisch, Japanisch, Koreanisch). Da muss man mit wenigen Tastendrücken aus 50000 Schriftzeichen auswählen können. Das funktioniert ähnlich wie das SMSen auf Handys: Der Benutzer gibt für jede Silbe eine Art Lautschrift ein, vorauf eine Liste mit Vorschlägen an der Cursorposition erscheint, aus der er das gewünschte Schriftzeichen auswählt.

Die Integration von Inputmethoden in Linux ist noch nicht ausgereift. Zum einen sind programmübergreifende Inputmethoden oft nicht einfach zu installieren. Andere gute, ergonomische Inputmethoden wie zum Beispiel die von Emacs oder Yudit ([6], Abbildung 4) sind nicht auf den gesamten Desktop anwendbar. Zum anderen wird immer nur eine Sprache oder Sprachfamilie auf einmal unterstützt. Nirgends gibt es beispielsweise eine Inputmethode für Ungarisch und Japanisch gleichzeitig. Die OpenI18N-Organisation arbeitet an einer weiteren, verallgemeinerten Inputmethode, die die alte X Input Method (XIM) ablösen soll [7]. ▶

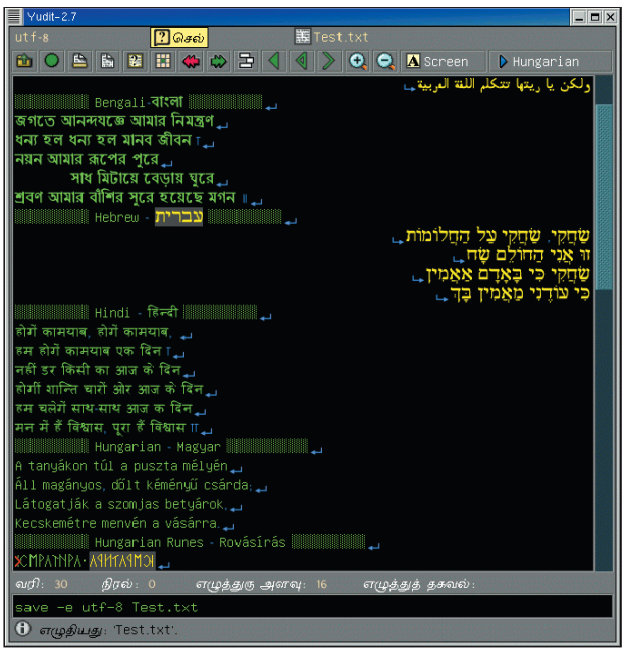

Abbildung 4: Der multilinguale Editor Yudit stellt im Gegensatz zu Firefox hebräische Schrift rechtsbündig dar.

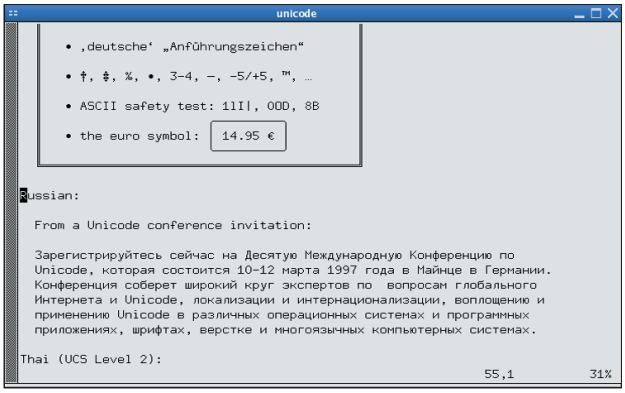

Abbildung 5: Die Unicode-Beispieldatei »UTF-8-demo.txt« im Editor Vim in einem Xterm: Sind die nötigen Fonts installiert, zeigen die Programme im Zusammenspiel die Schriften richtig an.

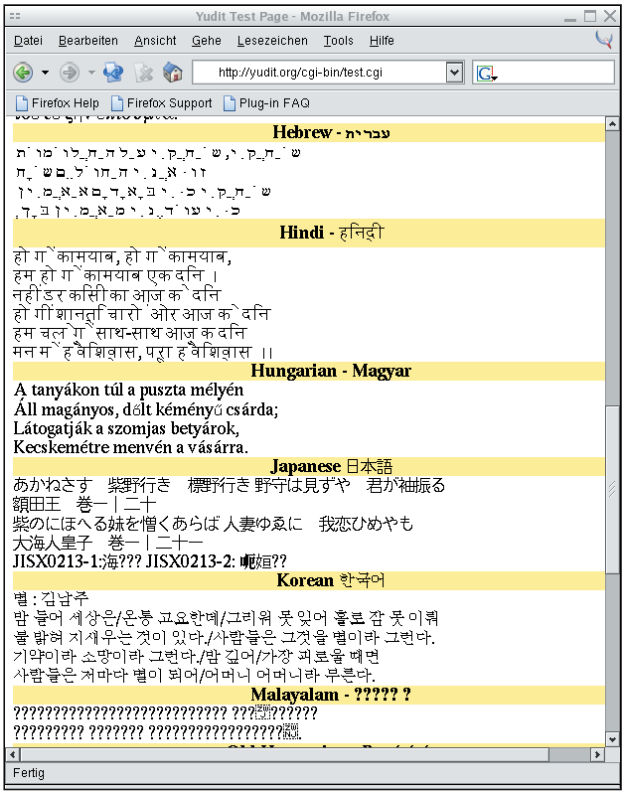

Abbildung 6: Firefox zeigt Webseiten mit Unicode an, wenn auch mit kleinen Macken. Fehlen Schriften, zeigt der Browser statt des richtigen Zeichens ein Fragezeichen, auch sonst ist das Rendering nicht optimal.

Darüber, ob der Standard seinen Anspruch eingelöst hat, gehen die Meinungen auseinander. Kritiker behaupten, viel Klarheit und Konsequenz der ursprünglichen Ideen sei zugunsten der Rückwärtskompatibilität aufgegeben worden. Speziell die in Unicode so genannte Round Trip Compatibility hätte zu schlechten Kompromissen geführt: Wird ein Zeichen von einem alten Encoding-System in Unicode umgewandelt und wieder zurück, muss das Ergebnis gleich dem Ausgangszeichen sein. Klar, dass Ungereimtheiten alter Encodings so in Unicode mit einfließen.

Das führt zum Beispiel dazu, dass für ein und dasselbe Zeichen in der realen Welt mehrere Unicode-Zeichen existieren, nur um die Round Trip Compatibility zu gewährleisten. Die Anwendungssoftware muss solche Mehrdeutigkeiten dann wieder ausgleichen.

Linux ist fast so weit

Wer aktuelle Distributionen einsetzt, findet eine Entwicklung im Übergang vor. Die Internationalisierung über UTF-8 hat lange begonnen, ist aber noch nicht abgeschlossen. Zu viele Komponenten müssen zusammenspielen, damit das Unicode-Dokument auf Bildschirm oder Papier erscheint.

Einzelne Anwendungen sind teilweise internationalisiert, andere überhaupt nicht, je nach verwendeten Bibliotheken und Toolkits (siehe Abbildungen 5 und 6). Wieder andere sind fast perfekt internationalisiert, arbeiten aber kaum mit anderen Programmen zusammen. Da die aktuellen GUIs und Desktops aber mittlerweile Unicode gut unterstützen, ist das nur noch eine Frage der Zeit. Der reibungslose Ablauf von der Eingabe über die Bildschirmdarstellung bis zum hochwertigen Ausdruck scheint nicht mehr fern. ■

Infos

[1] Unicode-Konsortium: [http://www.unicode.org]
[2] Cyberbit Unicode-Font: [http://titus.fkidg1.uni-frankfurt.de/unicode/tituut.asp]
[3] Unicode-Fonts Gentium und Doulos: [http://www.sil.org/computing/catalog/show_software_catalog.asp?by=cat&name=Font]
[4] Unicode-Normalisierungsformen: [http://www.unicode.org/unicode/reports/tr15/]
[5] IBM International Components for Unicode: [http://oss.software.ibm.com/icu/]
[6] Yudit-Editor: [http://www.yudit.org/]
[7] Input Methods von OpenI18N: [http://www.openi18n.org/modules.php?op=modload&name=Sections&file=index&req=viewarticle&artid=30&page=1]
[8] UTF-8 und Unicode-FAQ für Unix/Linux: [http://www.cl.cam.ac.uk/~mgk25/unicode.html]
[9] A Quick Primer On Unicode and Software Internationalization Under Linux and Unix: [http://eyegene.ophthy.med.umich.edu/unicode/]

Der Autor

Bruno Haible ist Autor des UTF8-Howto und Maintainer der Libiconv, die zwischen verschiedenen Zeichen-Encoding-Systemen konvertiert.

Mit Unicode programmieren

Für den Programmierer gibt es im Vergleich zu Ascii zwei Neuerungen: Er muss genau unterscheiden zwischen

- der Anzahl Bytes im Speicher, die ein String belegt,
- der Anzahl Unicode-Zeichen, die ein String enthält, und
- der Anzahl Spalten, die der String bei der Darstellung mit einem nicht-proportionalen Font beansprucht.

Beispielsweise ein Thai-Buchstabe mit Intonierungszeichen: Beide zusammen belegen 6 Bytes im Speicher, sind zwei Unicode-Zeichen und werden bei der Ausgabe in einer Spalte dargestellt.

Viele String-Operationen sind nicht mehr auf die einzelnen Zeichen anwendbar, sondern müssen den String als Ganzes betrachten. Selbst eine Umwandlung von Kleinbuchstaben in Großbuchstaben funktioniert nicht mehr eins zu eins, weil dabei beispielsweise Soße auf SOSSE abgebildet wird – ein Zeichen länger. Die Rückabbildung von Groß- zu Kleinbuchstaben liefert auch nicht das Original.

In mehreren Stufen zu I18N

Mit der Umstellung auf Unicode ist der erste Schritt für die Internationalisierung einer Software gegeben. Der zweite Schritt besteht darin, internationalisierbare APIs zu benutzen: »fprintf()« statt »puts()« und »strcat()« (weil beim Übersetzen manchmal ein Satz umgestellt werden muss); »gettext()« statt Strings, die nur in einer Sprache erscheinen können; »nl_langinfo()« für verschiedene Lokale-Abhängigkeiten. Der dritte Schritt, nunmehr in den Händen der Übersetzer, ist die Lokalisierung in verschiedene Sprachen.

Die Textprogrammierung ist durch Unicode um einiges komplexer geworden, sodass sich dafür statt C eine Hochsprache mit einer eigenständigen String-Klasse anbietet, etwa Java, Lisp, Perl oder Python. Für C(++) empfiehlt es sich, fertige Bibliotheken wie Pango, QT oder ICU zu verwenden.