Jörg Schilling

Die Geschichte

von

Autoconf & Co.

Fokus Fraunhofer

Was ist der Inhalt dieses Vortrags?

- Ein Überblick über die Anfänge der Portabilität
- Ein Überblick über Entwicklung von Build-Systemen
- Autoconf
- „Automake“
- Ein Überblick auf „nmake“ v. Glenn Fowler & David Korn
- Ein Überblick auf das Schily Makefilesystem

Warum wird Portabilität benötigt?

- Seit ca. 1978 gibt es Unterschiede in UNIX Versionen
  - Das entstand initial durch die Einführung von BSD UNIX
  - Dann kamen diverse kommerzielle UNIXe
  - Später Linux, MacOS X, … die nicht auf UNIX basieren
- Auch der POSIX Standard konnte das nicht verhindern
  - Einfache Programme sind durch POSIX abgedeckt
  - ...wenn sich die Plattformen an POSIX halten
  - Bei Linux hält man sich ungern an Regeln – auch nicht an Eigene
  - POSIX deckt auch nicht alle Eigenschaften ab
- Es gibt Plattformen, komplett jenseits von POSIX
  - z.B. Microsoft

Die Anfänge der Portabilität

- 1979: Das usenet entsteht als Verbindung zweier UNIX Rechner durch Kopplung mittels UUCP
  - Das usenet ist ein Versuch v. Hobbyisten etwas dem Arpanet vergleichbares entgegenzustellen
- Die Software dazu begründet die OpenSource Bewegung
- z.B. Der Newsreader „rn“ von Larry Wall
- 1980er Jahre: UNIX Versionen beginnen zu divergieren
  - Die gleiche Software kompiliert nicht mehr überall
- Larry Wall schreibt die erste Portabilitätshilfe
  - ...als handgeschriebenes Shell Skript

Was ist portable Software?

- Software, die auf mehreren Plattformen kompiliert
- Software, die auf mehreren Plattformen lauffähig ist
- Software, die zu den Eigenschaften der Plattformen paßt
  - Fehlende Eigenschaften der Plattform bei Bedarf ignoriert
  - Oder fehlende Eigenschaften bei Bedarf emuliert
- Software, die dazu an die aktuelle Plattform angepaßt werden kann
  - Dies kann manuell geschehen
  - Dies kann automatisiert sein

Was tut ein portables Build-System?

- **Anpassung an die Eigenschaften der aktuellen Plattform:**
  - **An verfügbare #include Files Anpassen**
  - **Bugs in System-#include Files umgehen**
  - **An verfügbare Bibliotheken anpassen**
  - **Eigenschaften der System-Bibliotheken beschreiben**
  - **Bugs in System-Bibliotheken umgehen**
  - **An Eigenschaften des Kompilers anpassen**
  - **Kommandozeile von Kompiler und Linker aufsetzen**
  - **Namensregeln für Bibliotheken o.ä.**

Was ist das Ziel optimaler Portabilität?

- Ein automatisches Build-System nimmt alles ab
- Möglichst nur die Namen der Quelldateien auflisten
- Das Build System sollte alles Sonstige wissen
- Verhindern nicht-portabler Dinge in den Makefiles
- Automatische Anpassung an bestehende und zukünftige Plattformen
- Verhindern von dupliziertem Code

Zurück zu den Anfängen

- Die Anfänge von OpenSource Software liegen im Usenet
  - Die Usenet Software selbst ist OpenSource
  - Das Usenet war d. erste Plattform z. Verteilen v. OSS
- OpenSource ohne Systematische Portabilität wäre tot
- Also zurück zum Usenet...

Ursprünge portabler Software und OSS

- 1979 wurde das USENET durch Tom Truscott und Jim Ellis gegründet und Software dazu als OSS verteilt
  - A-News durch Steve Daniel und Tom Truscott
  - B-News durch Matt Glickman und Mark Horton (Mary Ann H.)
  - C-News ist aktuell
- Die USENET Software entstand auf UNIX, wurde aber für viele andere Plattformen portiert
  - Leute wie Larry Wall, Rich Salz, Henry Spencer, Geoff Collyer haben die Grundlagen zur systematischen Portierung gelegt
  - Spätere Systeme haben Grundideen übernommen

Von der Ad Hoc Portierung zur Systematik

- **Ursprünglich hat der Nutzer Dateien manuell konfiguriert**
  - **Typischerweise wurden die Makefiles dazu manuell editiert**
- **Um 1980 gab es Ad Hoc Konfiguration (manuell)**
  - **Typische Konstrukte sind `#ifdef USG` (AT&T UNIX) UNIX Systems Group / UNIX System Laboratories**
  - **Vorteile: Anscheinend einfache Methode**
  - **Nachteile: Unflexibel zu Varianten in einer Plattform**
- **Um 1985 kam Larry Wall mit feingranularer Steuerung**
  - **Feature spezifisch: Typische Konstrukte sind `#ifdef TERMIO`**
  - **Vorteile: Varianten in einer Plattform beherrschbar**
  - **Nachteile: Viel Arbeit bei manueller Konfiguration**

Erste Ansätze zur Automatisierung

- Ende der 1980er: Larry Wall baut Mischung aus Automat. und manueller Eingabe für die Konfiguration
- Einige Dinge werden empirisch ermittelt, mit typischen Ausgaben als Meldungen:
  - „Looks kind of like a USG system, but we'll see..."
  - „Your System appears to use index() instead of strchr()“
- Andere Spezifika werden erfragt, typische Fragen:
  - „Is your 'test' built into sh?“
  - „Does your mailer understand Internet Addresses?“

- Um 1985: „imake“ entstand im Rahmen von X11
- Um 1992: GNU „autoconf“
- Um 1993: Das „Schily Makefilesystem“
- Um 1994: nmake + iffe von David Korn und Glenn Fowler
  - Iffe → If feature exists
- Um 1994: GNU „automake“, heute meist mit libtool
- Um 1996: GNU libtool
- Um 2000: „cmake“

Neuere Systeme müssen nicht zwangsläufig besser sein

imake: X11 ab ca. 1985

- Makefile-Präprozessor unter Verwendung von „cpp“
- Inputfiles: „Imakefile“ in cpp (C-Präprozessor) Syntax
- Outputfiles: „Makefile“
- Automatisches #include von „Imake.tmpl“
- Plattformanbieter editiert manuell Konfigurationsdateien
- Nachteile:
  - Nur der Umfang von X11 ist durch imake abgedeckt
  - Portabilität hängt vom Imakefile / Imake.tmpl Autor und v. System ab
  - Imake->make->make install->wegwerfen
  - Tot, seit X11 Release 7 (2004), das mit autoconf arbeitet

„autoconf“: GNU

- 1992 baut David McKenzie (GNU) „autoconf“
  - autoconf ist eine vollautomatisierte Version der Konzepte von Larry Wall
  - Der m4 Makroprozessor baut automatische Shell Tests
  - Jeder Test ist parameterisierbar
  - Ergebnis: ein riesiges Shell Skript „configure“
- „configure“ wird auf dem Zielsystem ausgeführt
  - Ergebnis: Ein File „config.h“ steuert die Kompilation
  - Typische Konstrukte: `#ifdef HAVE_TERMIOS_H`

Typischer autoconf Ablauf

- Der C-Kompiler wird auf Funktion geprüft, sonst Exit
- Einige Basis-Optionen vom Kompiler werden geprüft
- Das Vorhandensein diverser #include Dateien testen
- Das Vorhandensein diverser Funktionen in libc testen
- Komplexere Spezialtests

„autoconf“ Vor- und Nachteile

- Vorteil: Kommt theoretisch auch mit unbekannten Platt-formen klar
  - De-fakto klappt das wegen config.guess nicht
- Nachteile: configure->make->make install->wegwerfen
  - Also nicht besser als das imake Konzept

Weitere „autoconf“ Nachteile

- Viele Tests notwendig um „alles“ abzudecken (um die 1000)
- M4 Makrosystem schwer verständlich
- Langsam (weil sequentiell)
- Portabilität hängt auch vom Makefile.in Autor ab
- Nur durch Tricks gleichzeitige Kompilation für unterschiedliche Zielplattformen im selben Baum möglich
- Keine komplette Steuerung durch „make“
- Keine #include Dependencies → lückenhafte Make Regeln
- Keine lib Dependencies → lückenhafte Make Regeln
- Keine echte Unterstützung für Cross-Kompilation

Weitere „autoconf“ Nachteile

- „Ergebnisse“ d. Zielsystem Spezifika werden manuell erzeugt
- Ein Shell Skript der Größe von „configure“ könnte auch gleich „make“ ersetzen
- Keine echte Abstraktion
  - Die Doku schlägt vor, bestimmte Phrasen zu kopieren
  - Also Cut/Paste statt Referenz
  - Damit müssen viele Stellen gesucht / korrigiert werden, falls sich so ein Vorschlag als falsch herausstellt

„automake“: GNU

- GNU „automake“ stammt auch von David McKenzie
- „automake“ ist ein Versuch von GNU autoconf zu abstrahieren
- Im Idealfall listet ein Makefile.am nur die Quelldateien
- „automake“ ist ein Präprozessor für „autoconf“
  - Produziert *.in Dateien aus *.am Dateien
- „automake“ basiert auf einem Perl Skript

- Nachteile:
  - „GNU automake“ ist eine weitere Präprocessor Stufe
  - Nicht portabler Input führt zu nicht portablem Output
  - d.h. die Portabilität hängt vom Makefile.am Autor ab
  - GNU „automake“ kann nicht v. Linker abstrahieren
    - Speziell nicht von diversen Shared lib Konzepten
    - Dafür gibt es allerdings „libtool“
  - GNU „automake“ erfüllt nicht die Hoffunungen, die der Name „automake“ verspricht

- GNU „libtool“ stammt von Gordon Matzigkeit (1996)
- „libtool“ ist ein Shell Skript zur Abstraktion vom Dynamischen Linken
- Ziel ist eine einheitliche Kommandozeile unabhängig von der Zielplattform
  - Dadurch sollen Systemspezifika versteckt werden
- Nachteile: „libtool“ hat einen sehr schlechten Ruf, weil es erst Anfang 2010 – nach 14 Jahren - benutzbar wurde
  - Davor funktionierte es nur auf Linux
  - Damit hat es die Portabilität massiv verschlechtert

Libtool weitere Nachteile

- Die mangelnde Portabilität vor 2010 wirkt evt. noch Heute für Pakete, die nicht seit 2010 mit neuerem libtool verteilt wurden
- Dadurch haben z.B. FreeBSD und Solaris große Probleme beim Umgang mit OSS
- Libtool verlangsamt den Link Prozess merklich

„nmake“: AT&T

- Durch Glenn Fowler ab 1984 als „Dritte Generation make“ aus dem AT&T UNIX make Programm entwickelt
- Nmake „weiß“ wie man Code kompiliert
  - Nur die Quelldateinamen werden gelistet (wie beim Schily Makefilesystem)
- Seit 1994: automatische Konfiguration mit „iffe“
- Iffe ist ein Konfigurations-Interpreter als ksh-Skript
  - Iffe → „If Feature exists“, also ähnlich zu „configure“ v. autoconf
  - Viele #include Files aus „feature“ Files erzeugt
- AST-System → Advanced Software Technology, Basis v. ksh93
- Code des AST-Systems benötigt kaum #ifdef's
  - Das trifft aber auch auf das Schily Makefilesystem zu

„nmake“: AT&T

- Vorteile:
  - Elegante Arbeitsweise, wenn es bereits installiert ist
  - Hochportabel
- Nachteile:
  - Benötigt ein ksh93 Binary für den Bootstrap
    - Kann mit ksh-ähnlichem Shell umgangen werden
    - … bash ist also ein möglicher Bootstrap Helfer
  - Benötigt das üblicherweise fehlende „nmake“
    - Wird mit ksh93 Binary im Bootstrap kompiliert
  - Sehr gewöhnungsbedürftig

„cmake“: Kitware ab ca. 2000

- Ein Präprozessor für „CmakeLists.txt“ Dateien
- Autokonfiguration Interpreter ist in cmake eingebaut
- Die eigentliche Kompilation erfolgt mit „gmake“
- Interne Include Dependencies werden behandelt
- Interne Lib Dependencies werden behandelt
- Nachteile:
  - Eigentlich nur ein aufgepepptes veraltetes „imake“ Konzept
  - Benötigt „gmake“, aber „gmake“ läuft nicht auf allen Plattformen
  - Serielles Konfigurationskonzept wie bei „autoconf“
  - Code ist ohne eigene Anstrengungen nicht #ifdef-Frei
  - Kompilation/Editierung in unterschiedlichen Directories

Das „Schily Makefilesystem“

- Entwickelt ab Februar 1993, Solaris-x86 kam Dezember 1992
- Komplett automatisierte Steuerung mittels „make“
  - Regelgesteuert wird automatisch „autoconf“ aufgerufen
  - Danach wird Sub-Directory-Liste abgearbeitet
- Unterstützt aktuell „smake“, „SunPro-make“, „gmake“
  - Benötigte Zusatz-Features werden z.Zt. in POSIX standardisiert
- Make Regeln in separaten Platform-spezifischen Dateien
  - Kompilation wird in Quell-Directory gestartet
- Automatische Auswahl der Make Regeln mittels Pattern- Makro-Expansionen und „include“ Make-Direktive
- Gleichzeitige Kompilation für alle Zielplattformen im Baum

Das „Schily Makefilesystem“

- Einfache Makefiles enthalten nur Listen der Quelldateien
- Beispiel-Makefiles für diverse Target-Typen vorhanden
  - Makefiles für Sub-Directory-Listen
  - Makefiles für mehrere Sub-Targets
  - Makefiles für „Programme“
  - Makefiles für statische Biliotheken
  - Makefiles für dynamische Bibliotheken
  - ...
- Vorkonfiguriertes autoconf mit 900 Tests
  - ca. 10% davon müssen auf d. Zielplattform laufen
- Eingebaute Unterstützung für echte Cross-Kompilation

Das „Schily Makefilesystem“

- Enthält #include Files zur Abstraktion von der Plattform
  - z.B. #include <schily/stdio.h> statt #include <stdio.h>
  - Vermeidet so das copy & paste Chaos typischer autoconf Nutzer
  - Unterstützt so weitgehend #ifdef-freien Code
- Automatische Erzeugung der #include-Dependencies
  - Mit „smake“ geht das auch für cl.exe (Leerzeichen im Pfadnamen)
- Nachteile: Noch keine automatischen Lib-Dependencies
- Lib-Dependencies sind geplant (wie in SunPro Make)
- Verdeckte Erzeugung der #include-Dependencies ist geplant
- Wurde z.B. von der OpenLimit Ausweis-App verwendet

Das „Schily Makefilesystem“

- **Vorteile:**
  - **Extrem viele unterstützte Plattformen (> 30)**
  - **Automatische Bootstrap-Kompilation von „smake“ möglich**
  - **Einfache beliebige Erweiterbarkeit**
- **Nachteile:**
  - **Funktioniert am Besten mit smake / SunPro Make**
    - **Einige seltene Plattformen benötigen smake**
    - **...denn SunPro Make ist nur so portabel wie gmake**
  - **Benötigt ein Makefile pro Target**
    - **Mit einem SunPro Make Feature wäre das zu beheben**
  - **Gewöhungsbedürftig: ca. ein Tag Einarbeitungszeit**

Ein Beispiel-Makefile für „smake“

```
#ident @(#)Makefile     1.18 12/04/10
######################################################################################
SRCROOT=        ..
RULESDIR=       RULES
include         $(SRCROOT)/$(RULESDIR)/rules.top
######################################################################################

INSDIR=         bin
TARGET=         smake
CPPOPTS +=      -DUSE_LARGEFILES
CPPOPTS +=      -DNO_COMPAT '-DDEFAULTS_PATH="$(INS_BASE)/lib/defaults.smk"'
CPPOPTS +=      -DSCHILY_PRINT
CFILES=         make.c archconf.c readfile.c parse.c update.c rules.c \
                job.c memory.c
HFILES=         make.h job.h
#
# $(LIB_INTL) is needed because libschily/mem.c needs it.
#
LIBS=           -lschily $(LIB_INTL)
XMK_FILE=       Makefile.def Makefile.man

######################################################################################
include         $(SRCROOT)/$(RULESDIR)/rules.cmd
######################################################################################
```

Die Struktur des Schily Makefilesystems

`DEFAULTS/`	Directory mit Default Definitionen für bekannte Plattformen
`DEFAULTS_ENG/`	Definitionen für Entwickler (mehr Kompiler Warnungen)
`RULES/`	Die „make“-Regeln
`TARGETS/`	SSPM (Slottable Source Plugin Module) Directory: Steuerung sub-makes
`TEMPLATES/`	Beispiel-Makefiles
`autoconf/`	Das Schily „autoconf“ System, ein weiterentwickeltes GNU 2.13
`bins/`	Lokale Intermediate Binaries
`conf/`	Diverse Shell-Skripte
`inc/`	High-level autoconf
`include/`	Die Schily #include System-Abstraktion
`incs/`	Ergebnisse von „autoconf“ und den Programmen aus „inc“
`libs/`	Kompilations-Ergebnisse von lokalen Biliotheken

Das Schily Makefilesystem verstehen lernen

- http://sf.net/projects/schilytools/files Holen / kompilieren
- Enthält vollständigen „smake“ Bootstrap
  - Benötigt also kein vorinstalliertes „make“ auf dem Zielsystem
  - Daher auch auf Systemen nutzbar, auf denen „gmake“ nicht funktioniert
- Enthält eine Datei makefiles.tar.bz2
  - Darin ist eine minimale Version des Schily Makefilesystems
- Wenn smake vorhanden, makefiles.tar.bz2 in leerer Directory auspacken
- Danach kann das leere Makefilesystem erforscht werden

Das Schily Makefilesystem verstehen lernen

- Beim erstem Aufruf von smake wird die Autokonfiguration durchgeführt
  - Das sollte zunächst getan werden
- Aufruf von: `smake -xM` liefert Make-include Liste
  - Ausgabe ähnlich wie bei `cc -xM`
- Aufruf von: `smake -D` liefert ein expandiertes Makefile
  - Mit dem Inhalt aller Includes
  - Ausgabe ähnlich wie bei `cc -E`
- Für die weitere Erforschung empfiehlt es sich die Schily-tools anzusehen

Der Ablauf der initialen Kompilation

- Zu Beginn werden Symbolische Links angelegt
  - Oder Kopien, wenn auf aktueller Plattform nötig
- Die folgende Abfolge wird durch Einträge in der Directory „TARGETS“ gesteuert
  - Die Abarbeitung erfolgt alphabetisch
  - Die Nummerierung der Dateien erzwingt die nötige Ordnung
- Zuerst erfolgt der automatische „configure“ Lauf
- Danach weitere Autokonfigurations-Tests in Directory „inc“
- Danach wird die typischerweise mit 45... beginnende normale Targetliste abgearbeitet (ab 55... für normale Kommandos)

Eingliedern eines eigenen Programms

- Anlegen einer Directory im Verzeichnis-Root
  - z.B. `mkdir myprog`
- Danach Anlegen eines TARGET Eintrags
  - Durch `: > TARGETS/55myprog`
- Kopieren der Quelldateien in die Directory `myprog`
- Kopieren des Kommando-Template Makefiles
  - Durch `cp TEMPLATES/Makefile.cmd myprog/Makefile`
- Anpassen des Makefiles an `myprog`

Anpassen des Makefiles

- TARGET= auf unseren Namen myprog setzen
- CFILES= auf die Liste unserer Quelldateien setzen
- HFILES= auf die Liste unserer Include Dateien setzen
- LIBS= auf die Liste der benötigten Libs setzen
- Danach kann „smake“ aufgerufen werden
  - Unser Programm „myprog“ wird kompiliert
- Damit sollte es reichen für die Original-Plattform
- Für Portabilität muß das Programm auf das Portabilitäts Rahmenwerk angepaßt werden...

Warum „smake“?

- Ein portables Build-System benötigt ein portables make
- „Smake“ gibt es seit 1984, „gmake“ kam 1989
- Smake funktioniert auf mehr Plattformen als gmake
  - Gmake hat einen nicht POSIX-konformen Parser
  - Gmake behandelt „Leerzeichen“ falsch
  - Gmake unterstützt keine Leerzeichen in Dateinamen
  - Gmake behandelt Makefile-includes falsch
  - Gmake parallelisiert zu früh und hat kein Konzept zur gezielten Serialisierung
- Gmake-Bugs werden seit 1998 nicht beseitigt

Nachteile von smake

- Zur Zeit noch kein Support für Parallelisierung
- Support f. nebenläufige #include Dependencies würde Zeit sparen (siehe SunPro Make und environment Var SUNPRO_DEPENDENCIES)
- Support f. Library Dependencies (siehe SunPro Make und environment Var SGS_SUPPORT) würde die Kompilation zuverlässiger machen

- SunPro Make entstand 1986 als Rewrite vom UNIX make
- Viele neue Features:
  - Pattern matching Makro Ersetzungen
  - Pattern Matching Default Regeln
  - Bedingte Makro-Zuweisungen, abhängig vom aktuellen Target
  - Parallelisierung und verteiltes Arbeiten
  - Automatischer Support für #include Dependencies
  - Automatischer Support für lib Dependencies
  - OpenSource seit Dezember 2006
- Nachteile:
  - Binaries nur für Solaris und Linux/x86 (evt. HP-UX)
  - Schlecht gewartet (heute tot), neue Versionen nur alle paar Jahre

Originales OSS SunPro Make

- SunPro Make ist das Vorbild aller modernen Make Impl.
- Quellcode von SunPro Make ist seit Dezember 2006 verfügbar
- Ursprünglich nur mit den Eigenschaften von `/usr/ccs/bin/make`, daher:
- Kein Support für Parallelisierung und verteiltes Rechnen
  - Dieser Support wurde von Sun für OSS entfernt
- Erst einmal nicht portabel
  - Erkennbar ist rudimentärer Support f. Linux/HP-UX
- Verwendet seltsame nicht-portable Build-Umgebung

- Seit Anfang 2017 verfügbar – integriert in Schilytools
- Hochportabel (aber nicht so weitgehend wie smake)
- POSIX konform – Zertifiziert mit Solaris
- Erweitert um jüngste POSIX Features, z.B.:
  - „.PHONY:“ Special Target
  - „include“ kann nun mehrere Dateien auf Einmal
- Erweitert um Portabilitätssupport (eingebautes uname..)
- Paralellisierend arbeitend (verteilt aber n. nicht möglich)
- Unterstützt das Schily Makefilesystem vollständig

Wünsche für die Zukunft

- Parallelisierbare Autokonfiguration
- Automatische #include-Dependency Erzeugung bei der Kompilation mit Hilfe von SUNPRO_DEPENCENCIES=
- Automatische lib-Dependency Erzeugung mit Hilfe von SGS_SUPPORT=libmakestate.so oder LD_PRELOAD=
- Weitere Vorschläge?

Nützliche URLs

- http://www.kornshell.com/ Zum Download von AT&T
- http://www2.research.att.com/~gsf/man/ AT&T AST Man
- https://github.com/ksh93/ksh/ Vorsichtig gewarteter ksh
- http://sf.net/projects/schilytools/files Enthält aktuelles MF System
- http://cdrecord.org/Files/Geschichte_autoconf.pdf Dieser Vortrag
- http://sf.net/projects/schilytools/files/makefiles/PortableSoftware.ps.gz
-

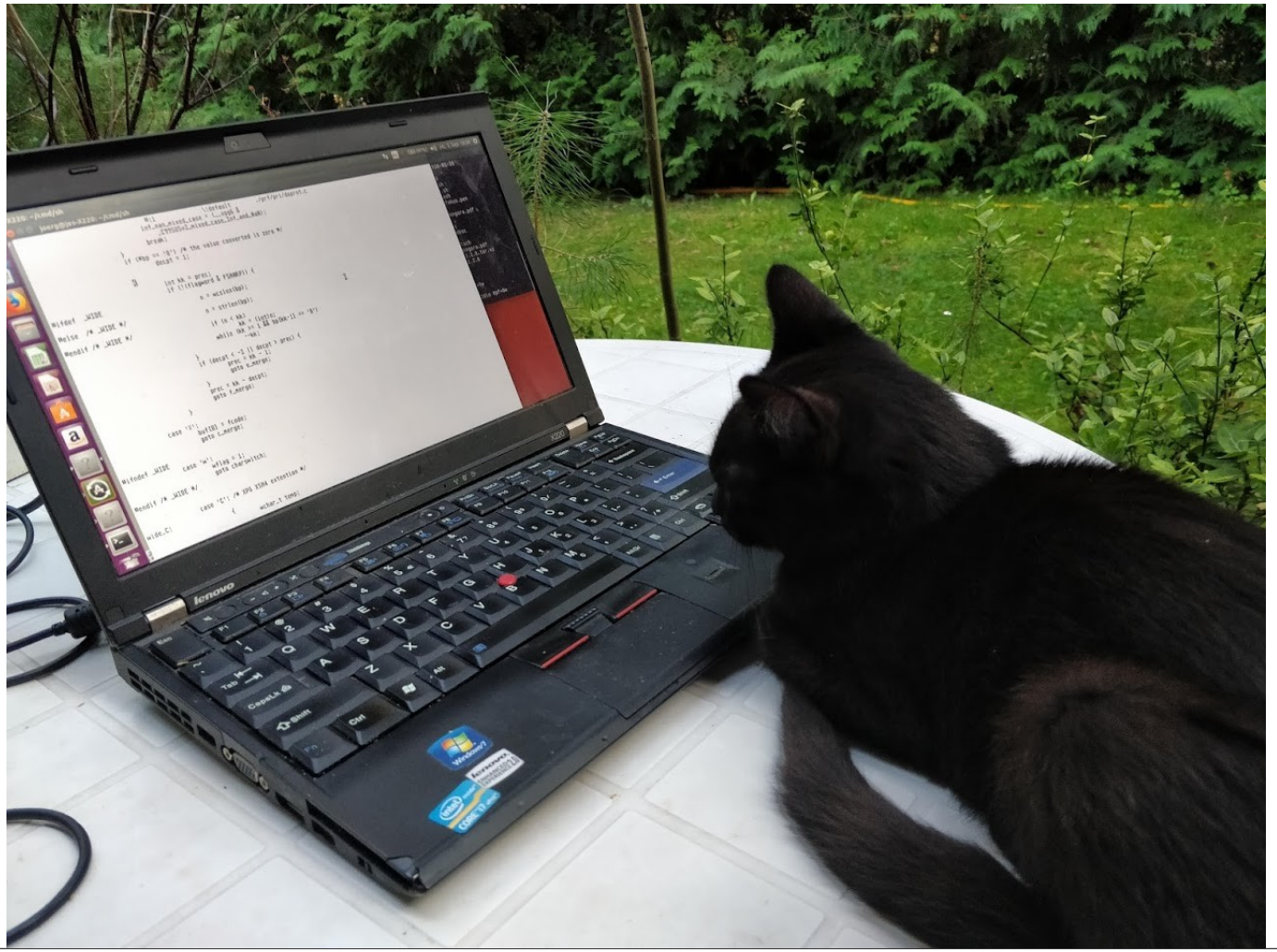
Jetzt noch etwas philosophisches

In zweifelhaften Fällen entscheide man sich für das Richtige

Karl Kraus

Neue Wege entstehen in dem wir sie gehen

Friedrich Mietzsche



Danke!