Programmentwicklung

Programmiersprachen

IDE (Integrated Development Environment)

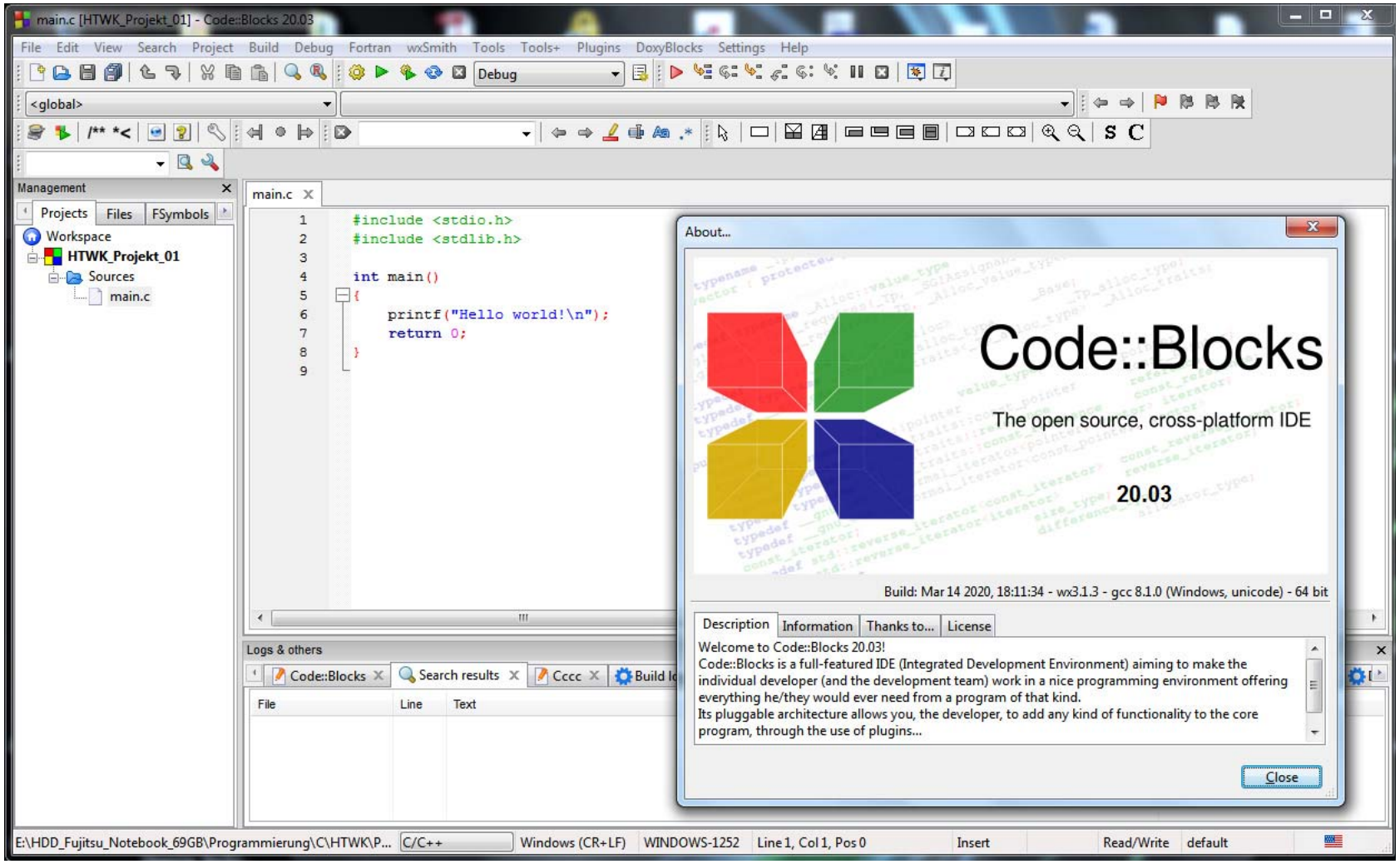
https://de.wikipedia.org/wiki/Liste_von_integrierten_Entwicklungsumgebungen

Code::Blocks (z.Z. Version 20.03 - Juni 2020)

http://www.codeblocks.org/
https://de.wikipedia.org/wiki/Code::Blocks
- freie, quelloffene Entwicklungsumgebung für C, C++, D und Fortran
- basiert auf wxWidgets
- läuft unter GNU/Linux, Windows und den meisten Unix-Derivaten
- Buildsystem ohne Makefiles
- unterstützt Syntax-Highlighting, Code-Faltung, Klassen-Browser
- Importfunktion für Visual Studio Projekte
- verschiedene Plugins

Orwell Dev-C++ (früher Dev-C++) (z.Z. Version 5.11 - Juni 2020)

https://sourceforge.net/projects/orwelldevcpp/
https://de.wikipedia.org/wiki/Orwell_Dev-C%2B%2B
- freie, quelloffene Entwicklungsumgebung für C, C++
- ursprünglich von Bloodshed Software
- nutzt MinGW als Compiler (eine Windows-Portierung der GNU Compiler Collection)


main.c [HTWK_Projekt_01] - Code::Blocks 20.03
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plugins DoxyBlocks Settings Help
Debug
<global>
Management
Projects Files FSymbols
Workspace
HTWK_Projekt_01
Sources
main.c
main.c
#include <stdio.h>
#include <stdlib.h>
int main()
{
printf("Hello world!\n");
return 0;
}
About...
Code::Blocks
The open source, cross-platform IDE
20.03
Build: Mar 14 2020, 18:11:34 - wx3.1.3 - gcc 8.1.0 (Windows, unicode) - 64 bit
Description Information Thanks to... License
Welcome to Code::Blocks 20.03!
Code::Blocks is a full-featured IDE (Integrated Development Environment) aiming to make the individual developer (and the development team) work in a nice programming environment offering everything he/they would ever need from a program of that kind.
Its pluggable architecture allows you, the developer, to add any kind of functionality to the core program, through the use of plugins...
Close
Logs & others
Code::Blocks
Search results
Cccc
File
Line
Text
E:\HDD_Fujitsu_Notebook_69GB\Programmierung\C\HTWK\P...
C/C++
Windows (CR+LF)
WINDOWS-1252
Line 1, Col 1, Pos 0
Insert
Read/Write
default

Datei Bearbeiten Ansicht Chronik Lesezeichen Extras Hilfe

The Dev-C++ Resource Site

www.bloodshed.net/dev/devcpp.html

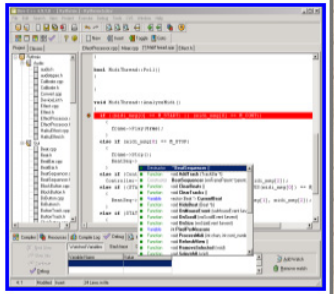
Suchen

The Dev-C++ Resource Site

Navigation:

- Home
- Dev-C++
- Packages
- Documentation
- Getting Help
- Credits
- Contact us

Feature list



- Support GCC-based compilers
- Integrated debugging (using GDB)
- Support for multiple languages (localization)
- Class Browser
- Code Completion
- Debug variable Browser
- Project Manager
- Customizable syntax highlighting editor
- Quickly create Windows, console, static libraries and DLLs
- Support of templates for creating your own project types
- Makefile creation
- Edit and compile Resource files
- Tool Manager
- Print support
- Find and replace facilities
- Package manager, for easy installation of add-on libraries
- CVS Support
- To-Do List
- CPU Window

Requirements

- Windows 95 or higher.
- 32 MB of RAM.
- The executables compiled by Dev-C++ will need MSVCRT.DLL (comes with Windows 95 OSR 2 or higher).

License

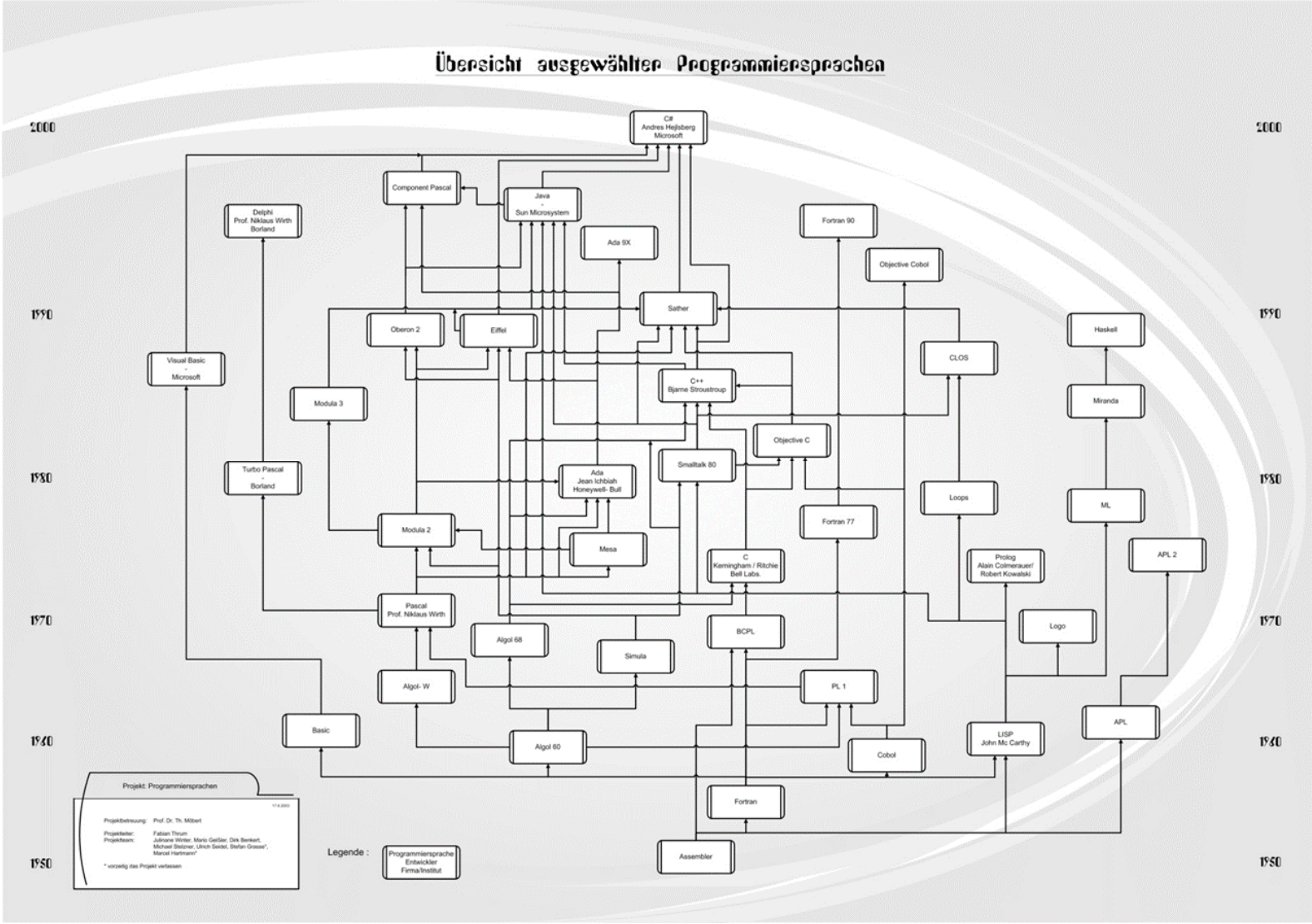
Dev-C++ is Free Software distributed under the GNU General Public License.
This means you are *free to distribute and modify Dev-C++*, unlike most Windows software! Be sure the read the license.

Donations

Please support Dev-C++ by making a donation ! The money will be shared between the active developers and the support manager in order to help us continue improving Dev-C++ from day to day.

Systematik der Programmiersprachen

1) imperative Sprachen (z. B) Basic o. ä.)
2) prozedurale Sprachen (z. B. ANSI-C oder Pascal)
3) objektorientierte Sprache (z.B. Smalltalk, CommonLisp, EIFFEL)
4) funktionale Sprachen (z. B. Haskell, ML, CommonLisp, Scheme, Dylan)
5) deklarativ-logische Sprachen (z. B. Prolog)
6) symbol-verarbeitende Sprachen (z. B. Scheme, CommonLisp)
7) Assembler-Sprachen (z. B. MIPS, x86, m68k)
8) Shell-Sprachen (z.B bash, csh, ksh, scsh...)
9) Markup-Sprachen (SGML, HTML...)
10) RDBMS-Query Sprachen (gibts eigentlich zur Zeit nur noch SQL)
11) Graphik/Seitenbeschreibungssprachen (PS=Postscript)
12) 3D-Szenenbeschreibungssprachen (VRML, OpenGL, PovRay, DXF)
13) Multimedia-Scripting (SMIL, LINGO, OpenScript)
14) Scriptsprachen (VBScript, JScript, Javascript, REXX, SAP ABAP)
15) Fachsprachen ...
16) Telekommunikationsskripte (Telix, Telemate)
17) Esoterische Programmiersprachen (Brainfuck, Whitespace, LOLCode, Piet, Ook!, HQ9+, ZOMBIE,BIT, Haifu, Whenever)


Übersicht ausgewählter Programmiersprachen
2000
1990
1980
1970
1960
1950
C#
Andres Heijlsberg
Microsoft
Component Pascal
Java
Sun Microsystem
Delphi
Prof. Niklaus Wirth
Borland
Ada 9X
Fortran 90
Objective Cobol
Sather
Oberon 2
Eiffel
Haskell
Visual Basic
Microsoft
CLOS
C++
Bjarne Stroustroup
Modula 3
Miranda
Objective C
Turbo Pascal
Borland
Smalltalk 80
Ada
Jean Ichbiah
Honeywell- Bull
Loops
ML
Modula 2
Fortran 77
Mesa
C
Kernigham / Ritchie
Bell Labs.
Prolog
Alain Colmerauer/
Robert Kowalski
APL 2
Pascal
Prof. Niklaus Wirth
Logo
Algol 68
BCPL
Simula
Algol- W
PL 1
Basic
APL
LISP
John Mc Carthy
Algol 60
Cobol
Fortran
Assembler
Projekt: Programmiersprachen
Projektbetreuung: Prof. Dr. Th. Möbert
Projektleiter: Fabian Thrum
Projektteam: Juliane Winter, Mario Geißler, Dirk Benkert, Michael Stelzner, Ulrich Seidel, Stefan Grosse*, Marcel Hartmann*
* vorzeitig das Projekt verlassen
Legende :
Programmiersprache
Entwickler
Firma/Institut

Programmiersprachengenerationen

1.Generation (Maschinensprachen)

- prozessorabhängig (z.B. x86)
- im Bootblock/ROM-BIOS
- kurze effiziente Programme
- verwenden absolute Adressen

2.Generation (Assemblersprachen)

- seit Anfang der 50er Jahre
- strukturäquivalent zu den Befehlen der Maschinensprache
- Zugriff auf die Prozessorbefehle (bzw. Peripherie)
- mnemotechnische Abkürzungen als Bezeichner für Variablen und Befehle
- Mechanismen zur Vereinfachung de Codierungsaufwandes
- Makros
- Übersetzung in Maschinenprogramme durch (Makro-)Assembler

3.Generation (prozedurale / problemorientierte Programmiersprachen)

- rechnerunabhängige Codierung prozeduraler Algorithmen
- Abstraktion von der Hard- und Softwarestruktur
- algorithmischer Aspekt hinsichtlich der Von-Neumann-Rechnerarchitektur

4.Generation (4GL / seit Ende der 60er Jahre)

- Definition und Operation auf Datenbank (SQL/NATURAL)
- Werkzeuge zur Gestaltung von Oberflächen (Masken)

5.Generation (deklarative Programmiersprachen)

- Problembeschreibung
- System
- logische Sprachen (PROLOG)
- funktionale Sprachen (ML, MIRANDA)

Kombination sinnvoll: logisch-funktionale Sprachen

Programmiersprachen - Konzepte

höhere Programmiersprachen

imperativ

Variablen = Speicherzellen im Rechner
d.h. Name → Adresse → Wert

Kennzeichen: - anweisungsorientiert
- explizite Darstellung des Steuerflusses im Sinne von Maschinenbefehlsfolgen eines J.v.Neumann-Rechners (Sequenzen,Alternativen,Sprünge, Selektionen,Iterationen,Rekursionen)
- Iterationen und Rekursionen können einander ersetzen

Bsp.: PL1,Fortran (**For**mular **Tran**slator), Algol,Pascal,
BASIC (**B**eginners **A**ll Purpuse**s** **I**nstru**c**tion Language)

applikativ

Variable = symbolischer Repräsentant eines Wertes

Kennzeichen:
- Kombination von Funktionen
- ohne explizite Darstellung des Steuerflusses
- im Vordergrund steht die Dekomposition des Gesamtproblems in einzelne Funktionen
- implizite (feste oder variable) Programmablaufsteuerung

funktional — *Bsp.*: LISP

logisch — PROLOG

objektorientierte Programmierung

Programmierung von Objekten mit speziellen Eigenschaften und Verhalten.
Die Objekte werden als Instanz einer Klasse (=gleichartige Objekte) erzeugt
und existieren dann eine gewisse Zeitdauer. Die einmal für eine Klasse beschriebenen
Eigenschaften und Verhaltensweisen (Methoden) sind jeder Instanz einer Klasse
zueigen und vererben sich in der aufgebauten Objekthierarchie von einer Superklasse
auf die Subklassen. Die Programmabarbeitung entspricht einem Nachrichtenaustausch
zwischen den Objekten und einer entsprechenden Reaktion und einer evtl.
Zustandsänderung des Objektes auf die empfangene 'message'. Datenabstraktionen
führen zur Möglichkeit der Verwendung verallgemeinerter Typen und Operatoren
sowie dynamischer Typprüfung. (RTTI = runtime type information)

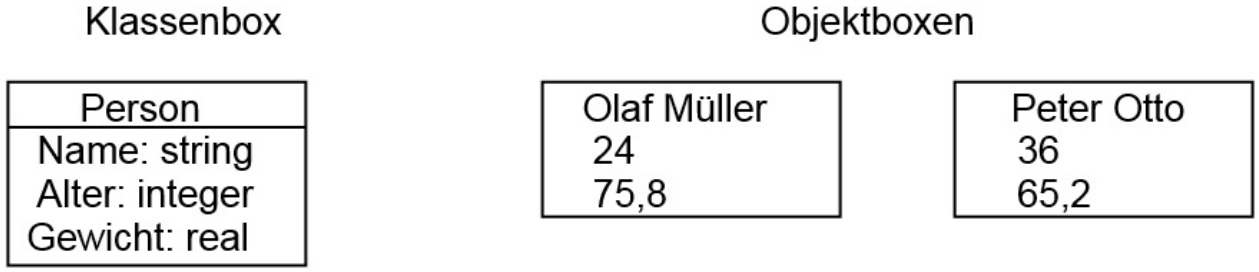
objektorientierte Programmierung

Kennzeichen: Objektorientierung / ereignisgesteuerte Programme

Abstraktion: Merkmalsreduktion; fundamentale menschliche Fähigkeit, welche es uns ermöglicht, die Komplexität der Realität zu erfassen und zu beschreiben

Objekt Modell: beschreibt die statische Struktur (Objekte, Beziehungen / objects, relationsships)
Dynamisches Modell: beschreibt die Steuer- (control) Struktur (Ereignisse, Zustände)
Funktions-Modell: beschreibt die Struktur der Werte (computational structure) – (Werte, Funktionen, Einschränkungen/values, functions, constraints)
Class-Diagramm: Schema, Pattern (Muster), Template (Struktur-Vorlage) zur Beschreibung endlich vieler Instanzen einer Klasse
Instanz-Diagramm: beschreibt eine Menge von Objekten, die in Beziehung zueinander stehen

Bsp: Klasse= Person
Instanzen der Klasse (Objekte): Olaf Müller, Peter Otto
Attribute: Name, Alter, Gewicht (Attribute haben nur Werte, aber keine existenzielle Identität)



objektorientierte Programmierung

Datenkapselung:
Zusammenschweißen von Programmteilen und Daten in einem Objekt

Polymorphie (griech. viele Formen, gestalten):
Einführung verschiedener Prozeduren (Methoden) unter gleichem Namen und gleicher Parameterliste

Vererbung:
Attribute und Methoden (=Prozeduren zur Manipulation von Objekten einer Klasse) können
auf Objekte der von der Oberklasse abgeleiteten Klasse vererbt (übernommen) werden.

Vorteile:
- einfache Erweiterbarkeit objektorientierter Entwürfe ohne bestehende Programmteile ändern zu müssen durch Hinzufügen des zusätzlichen Codes
- hochmoderne und angenehme Nutzerinterfaces
- späte Bindung = Linken erst während der Laufzeit (erfordert dynamische Link-Bibliothek)

Turbo Pascal

```
{$A+, B-,D+,E-,F+,I+,L+,N-,O+,R+,S+,V+}

(*$IFNDEF variable*){$M 50000,0,655360}
(*$ELSE*){$M 2000,0,655360} (*$ENDIF*)

PROGRAM programname;
USES OVERLAY,CRT,PRINTER, unitname,...;

{$O overlaymodul1}
{$O overlaymodul2}

(* Deklarationsteil des HP *)
CONST name1=...; name2=...;
TYPE  type1=...; type2=...;
VAR   name3:type1;
      name4:ARRAY[1..1000] OF type1;
LABEL 1,M1,name5,...;

FUNCTION fname(formalpar_liste):type;
(* Deklarationsteil der FUNCTION *)
CONST ...;
LABEL ...;
TYPE  ...;
VAR   ...;
BEGIN (* Anweisungsteil der FUNCTION *)
  ......................
END;
```

```
PROCEDURE procname1(formalpar_liste);
(* Deklarationsteil der PROCEDURE *)
CONST ...;
LABEL ...;
TYPE  ...;
VAR   name:...;
BEGIN (*Anweisungsteil der PROCEDURE*)
  ..................
  name:=funktionsname(...);
END;

PROCEDURE procname2(formalpar_liste);
(* Deklarationsteil der PROCEDURE *)
CONST ...;
LABEL ...;
TYPE  ...;
VAR   name:...;
 PROCEDURE geschachtelteProzedur;
 CONST ...;
 TYPE  ...;
 VAR   ...;
 LABEL ...;
 BEGIN (* Anweisungsteil *)
   ...............
 END;
BEGIN (* Anweisungsteil PROCEDURE 2 *)
  ..................
  name:=funktionsname(...);
END;
(*$I dateiname.pas*)
VAR ...;

BEGIN  (* Beginn des HP *)
  1:...;
M1:END.
```

Prolog

Prolog-Programm = Menge von Klauseln = Menge von Listen (eine Klausel wird als Liste notiert)

Liste = n-Tupel von Elementen (Objekten, Termen)

rekursive Def einer Liste:
() ist die leere Liste
(x|Z) ist eine Liste, falls Z eine Liste und x ein Element ist

Bsp:

```
((el Paul Horst))     ((w Ida))      ((alt Paul 1850 1901))
((el Paul Heinz))     ((w Ilse))     ((alt Ilse 1856 1918))
((el Paul Olga))      ((w Susi))     ((alt Horst 1876 1940))
((el Ilse Horst))     ((w Anna))     ((alt Heinz 1878 1941))
((el Ilse Heinz))     ((w Emma)      ((altOlga 1886 1964))
```

Zielklausel:

```
?((m Max)→liefert ?, weil ((m Max )) nicht als Fakt vorhanden
? ((el Paul z))   →  ohne Anzeige
? ((el Paul z) (PP z))   →  Horst
? ((el Paul z) (PP z) FAIL)    →  liefert   Horst
                                            Heinz
                                            Olga

WHICH (z (el Paul z))   →   liefert  Horst
                                     Heinz
                                     Olga
```

Programmiersprachen

"Hello-World" in Postscript

```
%!PS-Adobe-2.0 EPSF-2.0
%%DocumentFonts: Courier-Bold Helvetica-Bold
%%BoundingBox: 5 -5 540 120
%%Title: test.eps
%%EndComments

0 200 moveto
(Hallo Welt) show  % schwarz
gsave
   0 100 moveto
   1 0 0 setrgbcolor
   (hallo Welt) show  % rot
grestore

showpage
```

Hello World Programm in C#

```
using System;
public class MyHelloWorld{
public MyHelloWorld() {
   console.WriteLine('Hello World!');
   }
}
```

HelloWorld in MSIL:

Microsoft Intermediate Language

```
.assembly MyAssembly {}
.class MyApp {
.method static void Main()
{ entrypoint
  ldstr "Hello World!"
  call void
  System.Console::WriteLine
    (class System.Object)
  ret
} }
```

HelloWorld in Turbo Pascal:

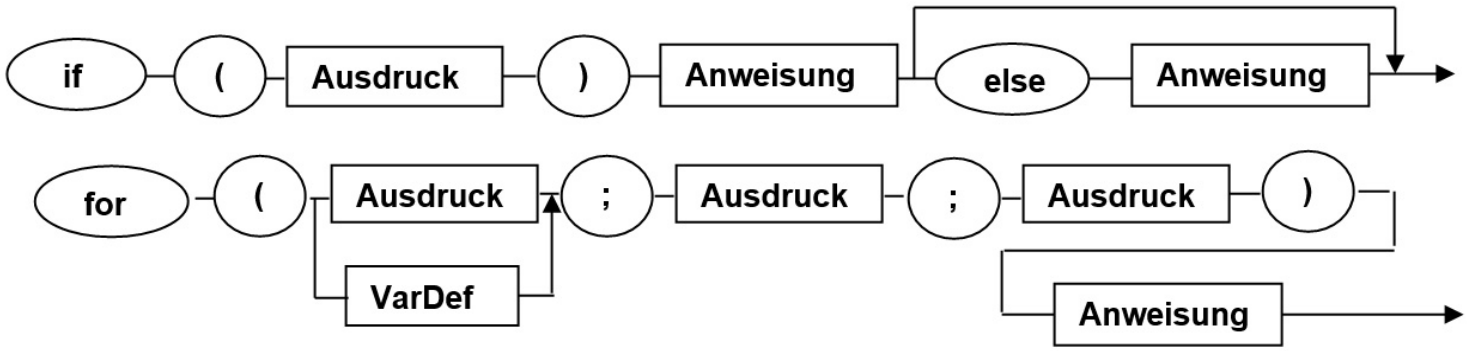
```
program test;
uses Overlay,CRT,DOS,TOOLS,ReadChr;
const b:byte=13;
var i:integer;r:real;

procedure ausschrift;
begin
   witeln('Hello world!');
end;

begin (* Beginn des Hauptprogrammes *)
    ausschrift;
    i:=5;r:=12,3456;
    writeln('i=', i:2,'   r=',r:6:2,chr(b));
end.
```

Syntaxdiagramme

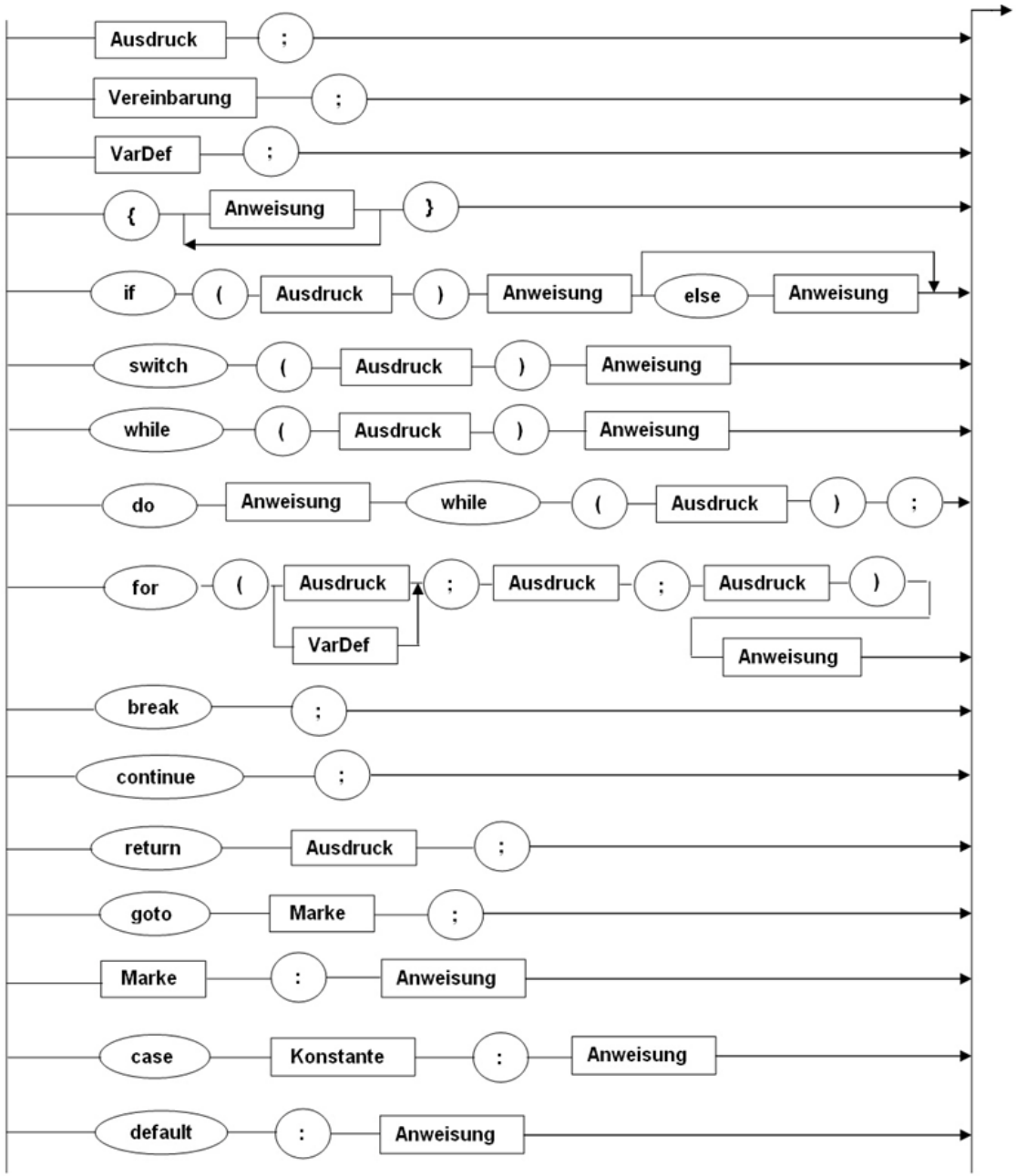
Beschreibung von Sprachen mittels graphischer Elemente (Knoten und gerichtete Bögen zwischen diesen Knoten). Jeder Weg in einem solchen Syntaxdiagramm beschreibt ein syntaktisch zulässiges Sprachkonstrukt der zu definierenden Sprache.

Sprachelemente:

Allgemeine Symbole: Programmablaufrichtung, Kommentar
Sequenz
Alternative (einseitig = nur if-Zweig, zweiseitig if+else-Zweig)
Mehrfachalternative
Abweisende Wiederholung (Zyklus)
Nichtabweisende Wiederholung (Zyklus)
Unterprogramm



Syntaxdiagramm der Programmier-sprache C



Formale Sprachen / Metasprachen

Def: künstliche Sprache = Formale Sprache (Lexik,Syntax) + Interpretation/Bedeutung (Semantik)

Def: Grammatik ist eine Beschreibung einer Sprache (Syntaxregeln), d.h. eine Metasprache, also eine Sprache zur Beschreibung einer Sprache.

Def: Ein Alphabet ist eine beliebige, endliche, nichtleere Menge.
Die Elemente eines Alphabetes heißen Zeichen oder Symbole.

Def: Ein Wort (über dem Alphabet A) ist eine Verkettung (Konkatenation/Hintereinanderschreibung) von endlich vielen Zeichen aus ein und demselben Alphabet A. ➔ vereinbarte Schreibweise: "abcdef"

Def: BNF (Backus Naur Form, vorher Backus-Normal-Form) ist eine formale Beschreibung einer Grammatik definiert durch Jim Backus/Pete Naur zur Definition Algol60

Weiterentwicklungen:
- EBNF (Extendend BNF) durch Nikolaus Wirth für Pascal
- ISO/IEC 14977 (EBNF)
- BS 6154
- ABNF (Augmented BNF nach RFC 2234)
- XML, YACC/BISON-Grammers, XBNF

Die erweiterte Backus-Naur-Notation (EBNF)

Diese Metasprache dient der Beschreibung Syntax einer Programmiersprache, Kommandosprache, Fachsprache oder sonstigen formalen Sprachen.

Die erweiterte Backus-Naur-Notation besteht aus den folgenden Sprachelementen:

a) **Notations- oder Metavariable** = Begriff, der nicht selbst Element der zu definierenden Sprache ist
(wird in Kleinbuchstaben notiert oder in Klammern <>)
Bsp.: (alle folgenden Beispiele beziehen sich auf die Beschreibung von MSDOS-Betriebssystemkommandos)
datei1 pfad dateiname laufwerk ..oder .. <datei1> <pfad> <dateiname> <laufwerk>

b) **Notationskonstante**= Zeichen oder Zeichenfolge der zu definierenden Sprache
Bsp.: DIR TYPE TREE FORMAT ERASE CD MD RD .....

c) **Definitionssymbol "::="** zur Definition von Metavariablen
Syntax: zu definierende Matavariable ::= Ausdruck in Backus-Form
Bsp.: kommando1::= DIR dateiname

d) **Alternativsymbol "|"** zur Trennung alternativer Notationskonstanten/Notationsvariablen
(als 'oder' zu lesen)
Bsp.: internes_Kommando::=DIR|TYPE|REN|DEL|TIME|DATE|CD|MD|RD

e) **Möglichkeitssymbol "[ ]"** ; die in den Klammern stehenden Metavariablen bzw. Notationskonstanten können an dieser Stelle stehen, müssen es aber nicht
Bsp.: DIR [/w /p dateispezifikation]

f) **Auswahlsymbol "{ }"** ; eine der vom Auswahlsymbol umschlossenen Alternativen muss stehen
Bsp.: COPY { datei | datei1 datei2 | datei pfadangabe }

g) **Wiederholungssymbol "*"** ; die vor dem Wiederholungssymbol stehende Metavariable/Konstante bzw. Metaausdruck kann mehrmals hintereinander stehen
Bsp.: vollständige_pfadangabe::={\knotenname}*
also z.B.: pfad::=\AAA\BBB\CCC\DDD

h) **Listensymbol "'°°°"** ; analog wie das Wiederholungssymbol, aber vor jeder Wiederholung muss ein Komma (bzw. ein speziell definierter Separator) stehen. Unter einer Liste versteht man eine durch Komma getrennte Aneinanderreihung von Sprachelementen

Definition eines deutschen Satzes:

```
<Satz> = <Subjekt> <Prädikat>[<Objekt>] .
<Subjekt> =  <Eigenname> | <Artikel><Substantiv>.
<Prädikat> = <Verb> .
<Objekt> = <Artikel><Substantiv>
```

Formeldefinition für die Grundrechenarten:

Start:
```
<formel> ::= <ausdruck>
     <ausdruck> ::= <term> { + <term> | - <term> }
     <term>     ::= <faktor> { * <faktor> | / <faktor> }
     <faktor>   ::= <zahl> | ( <ausdruck> )
     <zahl>     ::= [+|-] <ziffer> {<ziffer>}* [. {<ziffer>}*]
     <ziffer>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Definition einer "Paradiessprache":

```
<Eigenname> = "Adam" | "Eva" .
<Verb> = "ißt" | "beißt" | "liebt" .
<Artikel> = "der" | "die" .
<Substantiv> = "Apfel" | "Schlange" .
```

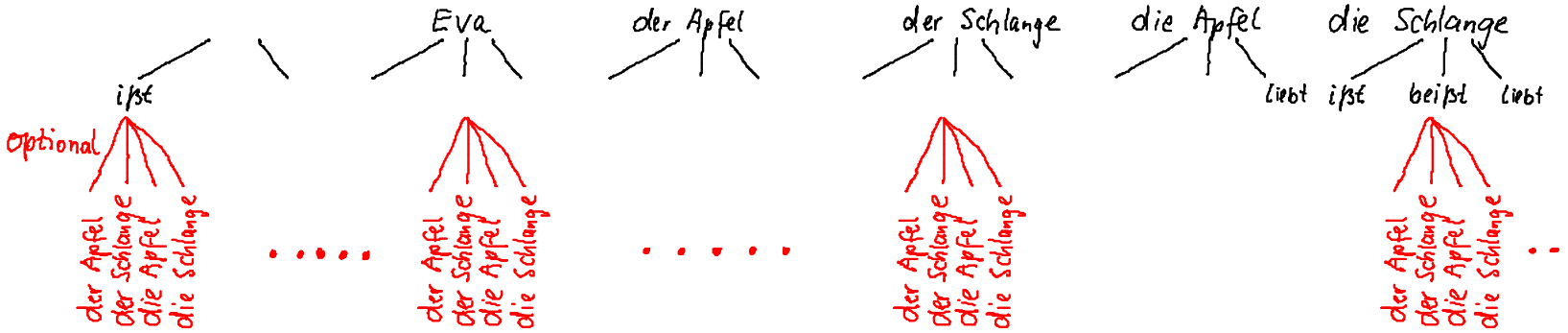
Bsp: Startsymbol: <paradiessprache> ::= (<satz>)*

<satz> ::= <subjekt> <prädikat> [<objekt>].
<subjekt> ::= <eigenname> | <artikel> <substantiv>
<prädikat> ::= <verb>
<objekt> ::= <artikel> <substantiv>
<eigenname> ::= "Adam" | "Eva"
<verb> ::= "ißt" | "beißt" | "liebt"
<artikel> ::= "der" | "die"
<substantiv> ::= "Apfel" | "Schlange"

(1. Regel, rule, Produktion)

zulässige Sätze (auch manchmal Worte genannt) der definierten Sprache

Bsp: Adam liebt Eva. ↯ kein zulässiger Satz
der Apfel beißt die Schlange.
Eva ißt.
⋮



Problem: (Quoting) falls ein Metasymbol selbst Bestandteil der zu definierenden Sprache ist, so muß die Meta-Bedeutung für dieses Zeichen aufgehoben werden (in der reduzierten Notation ohne "").Dabei benutzt man ein Quotingsymbol
Bsp: <sondertext> ::= Mengen werden in \[\] eingeschlossen und 2\>1. z.B. "\"

ABNF – Augmented Backus Naur Form – RFC 2234 → RFC 4234 → RFC 5234


```
        rulelist       =  1*( rule / (*c-wsp c-nl) )

        rule           =  rulename defined-as elements c-nl
                               ; continues if next line starts
                               ;  with white space

        rulename       =  ALPHA *(ALPHA / DIGIT / "-")

        defined-as     =  *c-wsp ("=" / "=/") *c-wsp
                               ; basic rules definition and
                               ;  incremental alternatives

        elements       =  alternation *c-wsp

        c-wsp          =  WSP / (c-nl WSP)

        c-nl           =  comment / CRLF
                               ; comment or newline

        comment        =  ";" *(WSP / VCHAR) CRLF

        alternation    =  concatenation
                          *(*c-wsp "/" *c-wsp concatenation)
```

```
concatenation  =  repetition *(1*c-wsp repetition)

repetition     =  [repeat] element

repeat         =  1*DIGIT / (*DIGIT "*" *DIGIT)

element        =  rulename / group / option /
                  char-val / num-val / prose-val

group          =  "(" *c-wsp alternation *c-wsp ")"

option         =  "[" *c-wsp alternation *c-wsp "]"

char-val       =  DQUOTE *(%x20-21 / %x23-7E) DQUOTE
                       ; quoted string of SP and VCHAR
                       ;  without DQUOTE

num-val        =  "%" (bin-val / dec-val / hex-val)

bin-val        =  "b" 1*BIT
                  [ 1*("." 1*BIT) / ("-" 1*BIT) ]
                       ; series of concatenated bit values
                       ;  or single ONEOF range

dec-val        =  "d" 1*DIGIT
                  [ 1*("." 1*DIGIT) / ("-" 1*DIGIT) ]

hex-val        =  "x" 1*HEXDIG
                  [ 1*("." 1*HEXDIG) / ("-" 1*HEXDIG) ]

prose-val      =  "<" *(%x20-3D / %x3F-7E) ">"
                       ; bracketed string of SP and VCHAR
                       ;  without angles
                       ; prose description, to be used as
                       ;  last resort
```

Syntaxbeschreibung mittels modifizierter BNF (RFC 822 / E-Mail-Format)

a) RULE1 / RULE2: ALTERNATIVES (Alternative)
b) (RULE1 RULE2): LOCAL ALTERNATIVES (lokale Alternative)
c) *RULE: REPETITION (Wiederholung)

`<l>*<m>element`

bedeutet: mindestens 1 element, höchtens m elemente
0 und unendlich sind Standard

Bsp: `1*DIGIT` bedeutet mindestens eine Ziffer, `*3DIGIT` bedeutet: höchtens 3 Digits

d) [RULE]: OPTIONAL (Möglichkeitssysmbol)
e) NRULE: SPECIFIC REPETITION (Wiederholung)

`"<n>(element)" is equivalent to "<n>*<n>(element)"`

bedeutet: genau n-faches Auftreten des Elementes

f) #RULE: LISTS

`<l>#<m>element`

bedeutet: mindestens 1 element, höchtens m elemente durch Komma getrennt

g); COMMENTS (Kommentar bis zum Zeilenende)
h) "\" (Quotingsymbol)

Aufheben der Sonderbedeutung eines Zeichens

i) "zeichenkette"

```
j) LEXICAL TOKENS                                  ; (  Octal, Decimal.)
     CHAR        =  <any ASCII character>          ; (  0-177,  0.-127.)
     ALPHA       =  <any ASCII alphabetic character>
                                                   ; (101-132, 65.- 90.)
                                                   ; (141-172, 97.-122.)
     DIGIT       =  <any ASCII decimal digit>      ; ( 60- 71, 48.- 57.)
     CTL         =  <any ASCII control             ; (  0- 37,  0.- 31.)
                    character and DEL>             ; (    177,     127.)
     CR          =  <ASCII CR, carriage return>    ; (     15,      13.)
     LF          =  <ASCII LF, linefeed>           ; (     12,      10.)
     SPACE       =  <ASCII SP, space>              ; (     40,      32.)
     HTAB        =  <ASCII HT, horizontal-tab>     ; (     11,       9.)
     <">         =  <ASCII quote mark>             ; (     42,      34.)
     CRLF        =  CR LF
     LWSP-char   =  SPACE / HTAB                   ; semantics = SPACE
     linear-white-space =  1*([CRLF] LWSP-char)    ; semantics = SPACE
                                                   ; CRLF => folding
     specials    =  "(" / ")" / "<" / ">" / "@"    ; Must be in quoted-
                 /  "," / ";" / ":" / "\" / <">    ;  string, to use
                 /  "." / "[" / "]"                ;  within a word.
     delimiters  =  specials / linear-white-space / comment
     text        =  <any CHAR, including bare      ; => atoms, specials,
                    CR & bare LF, but NOT          ;  comments and
                    including CRLF>                ;  quoted-strings are
                                                   ;  NOT recognized.
     atom        =  1*<any CHAR except specials, SPACE and CTLs>
     quoted-string = <"> *(qtext/quoted-pair) <">; Regular qtext or
                                                   ;   quoted chars.
     qtext       =  <any CHAR excepting <">,       ; => may be folded
                    "\" & CR, and including
                    linear-white-space>
     domain-literal =  "[" *(dtext / quoted-pair) "]"
     dtext       =  <any CHAR excluding "[",       ; => may be folded
                    "]", "\" & CR, & including
                    linear-white-space>
     comment     =  "(" *(ctext / quoted-pair / comment) ")"
     ctext       =  <any CHAR excluding "(",       ; => may be folded
                    ")", "\" & CR, & including
                    linear-white-space>
     quoted-pair =  "\" CHAR                       ; may quote any char
     phrase      =  1*word                         ; Sequence of words
     word        =  atom / quoted-string
```

Programmentwicklungszyklus

(compilierende Arbeitsweise)

1) algorithmischer und struktureller **Entwurf** (PAP, Struktogramm, UML-Diagramm)

2) Erarbeitung/Editieren des Quelltextes: **Implementation/Codierung**
des Programmes in einer Programmiersprache mit Hilfe eines Texteditors, evtl. auch mit Hilfe eines Quelltextgenerators, d.h. mittels CASE-Tool

3) Übersetzen des Quelltextes mittels eines Compilers:
Einpaß-bzw. Mehrpaßcompiler; evtl. mit Codeoptimierung;
Erzeugung des Maschinencodes [Objektcodedatei= *.OBJ]
mit provisorischen Einsprung- und Rücksprungadressen

4) Verbinden ("linken") des Programmes, d.h. Zusammensetzen der verschiedenen compilierten Module und der Laufzeitumgebung zu einem ausführbaren Lademodul (**"**binaries**"**) mit Hilfe des Linkers. Dabei findet eine Klärung der gegenseitigen Referenzen (Berechnung der Variablen- und Moduladressen) einschließlich der Anbindung der verwendeten Bibliotheksmodule statt. Außerdem wird ein für das jeweilige Betriebssystem genormter Programmheader erzeugt. Im Gegensatz zu diesem 'statischen Bindevorgang' werden beim 'dynamischen Binden' (= späte Bindung) die einzelnen Moduln erst zur Laufzeit aus den dynamischen Linkbibliotheken bzw. aus dem Hauptspeicher geladen und gebunden. Damit wird u.a. ein Code-Sharing durch mehrere Applikationen ermöglicht.

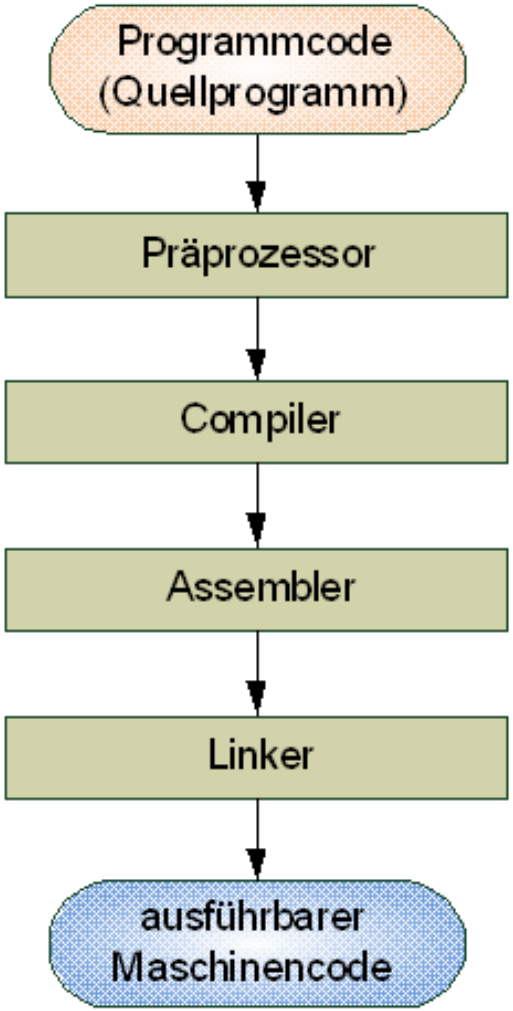
Programmentwicklungszyklus

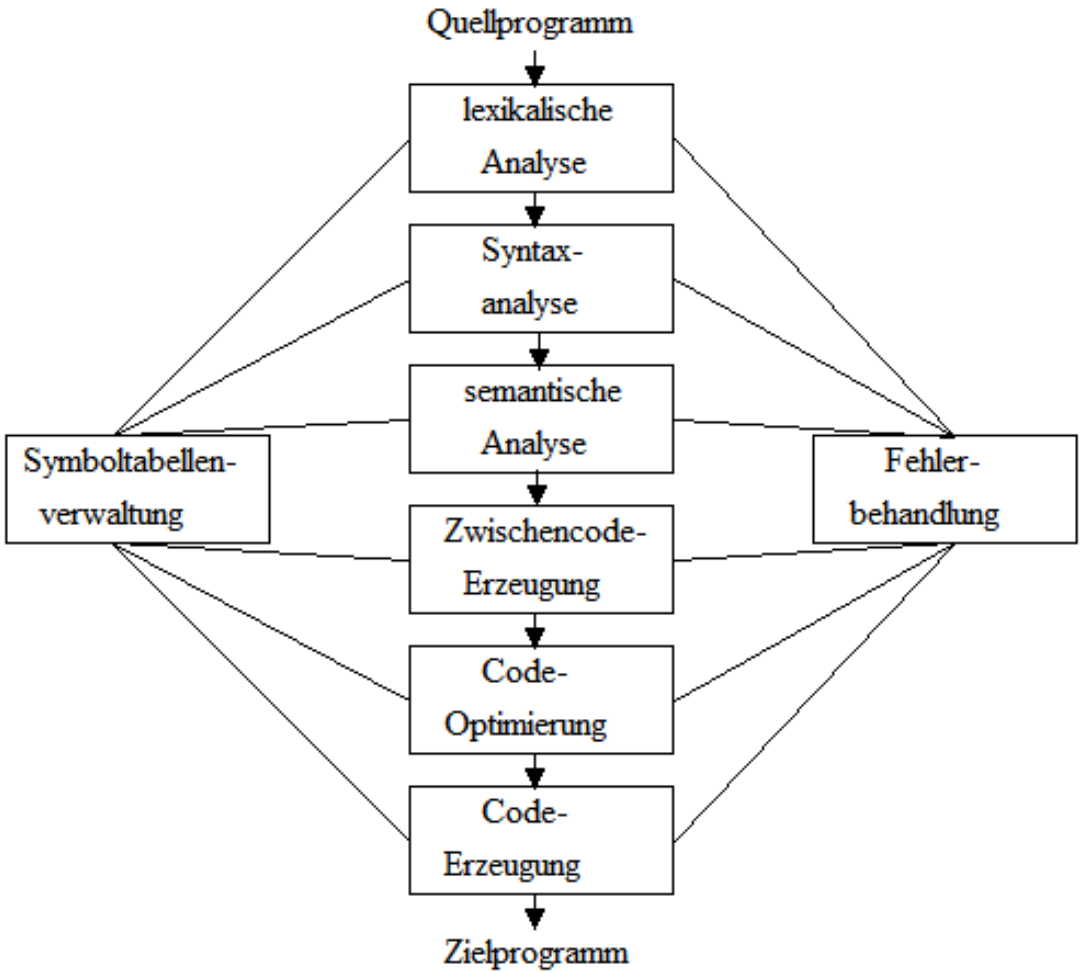
(compilierende Arbeitsweise)

5) Laden des Lademoduls in den Hauptspeicher und Start des Programmes
an dem "Entry Point" (Startadresse) ➔ Laufzeit, Prozess

6) Testen des Programmes z.B. mit Hilfe eines Debuggers
Hierbei wird der Lademodul schrittweise (z.B. zeilenweise "Traceing") abgearbeitet.
Es können Programmunterbrechungspunkte ("Breakpoints") gesetzt, momentane Variablenbelegungen und Modulaufrufverschachtelungen angezeigt sowie Prozessorregister und Variablenwerte verändert werden.
In "Watch-Points" kann man aktuellen Werte ausgewählter Variablen anzeigen.
Bei Quelltextdebuggern besteht die Möglichkeit des Verfolgens des Programmablaufes am Quelltext.

➔ bei einer interpretativen Arbeitsweise wird der Quelltext **"**zeilenweise**"**
mittels eines **Interpreters** übersetzt und sofort abgearbeitet


Programmcode
(Quellprogramm)
Präprozessor
Compiler
Assembler
Linker
ausführbarer
Maschinencode

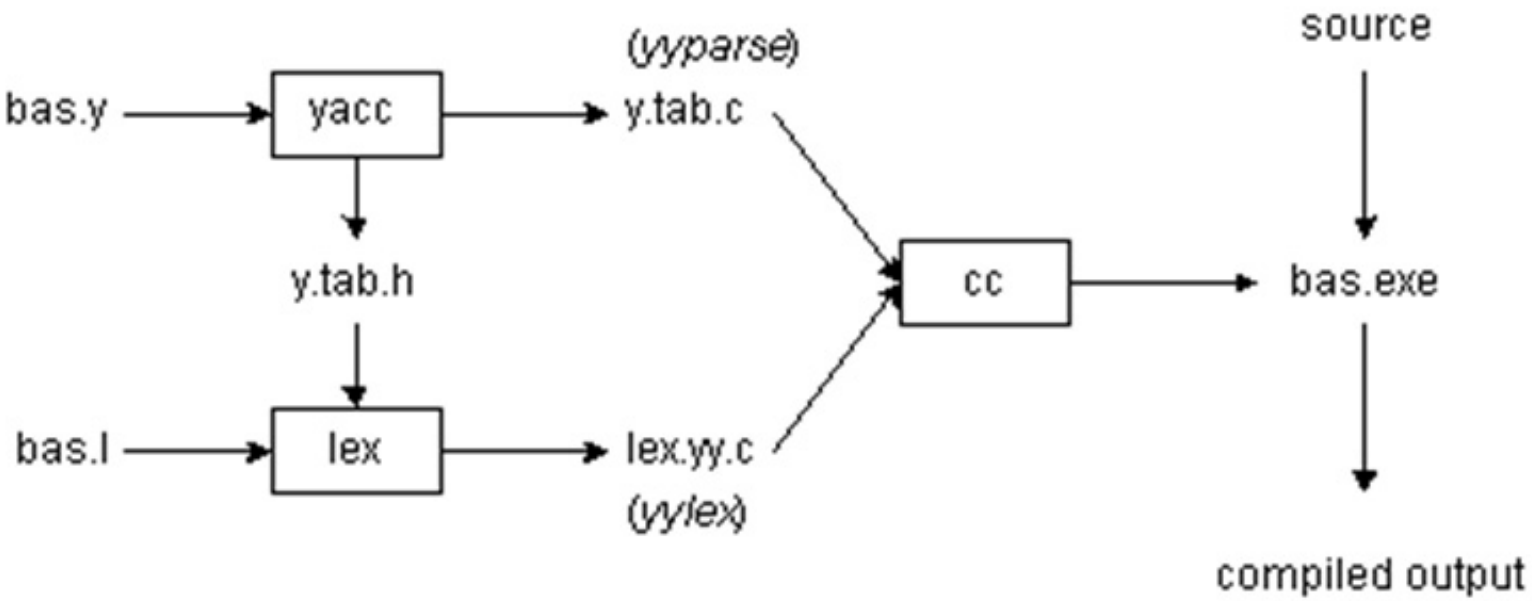

Quellprogramm
lexikalische
Analyse
Syntax-
analyse
semantische
Analyse
Zwischencode-
Erzeugung
Code-
Optimierung
Code-
Erzeugung
Zielprogramm
Symboltabellen-
verwaltung
Fehler-
behandlung

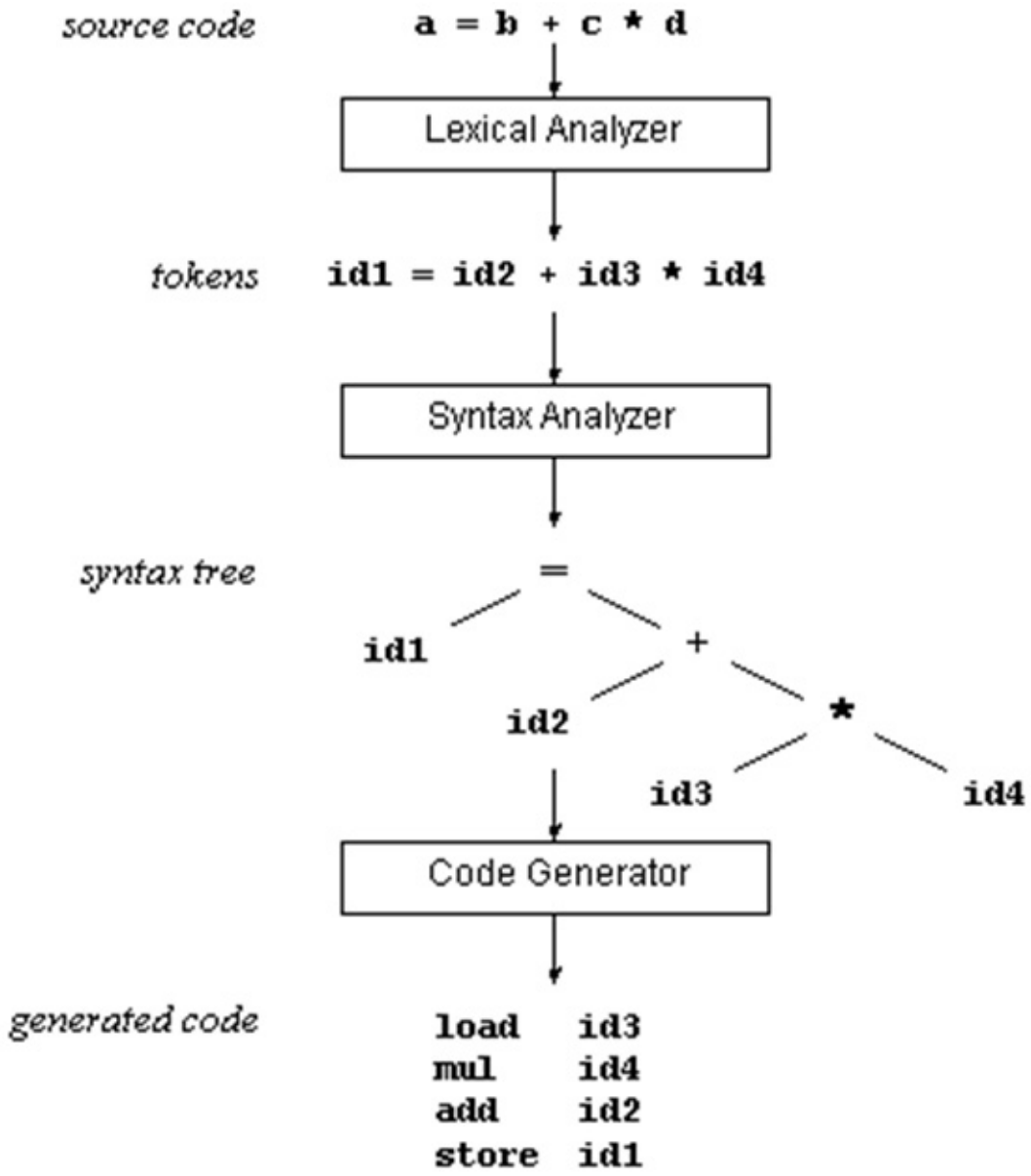
Parser/Compiler-Tools

Lex & YACC (siehe z.B. http://www.epaperpress.com/lexandyacc/index.html)

LEX - A Lexical Analyzer Generator (Lesk 1975)

YACC - Yet Another Compiler-Compiler (Johnson 1975)




source code
a = b + c * d
Lexical Analyzer
tokens
id1 = id2 + id3 * id4
Syntax Analyzer
syntax tree
=
id1
+
id2
*
id3
id4
Code Generator
generated code
load id3
mul id4
add id2
store id1

Programmentwicklungszyklus

(compilierende Arbeitsweise)

Laufzeitbibliothek:

Module zur Realisierung komplexer Sprachkonstrukte (math. Funktionen)
Systemfunktionen, die nicht vom Betriebssystem bereitgestellt werden
Fehlerbehandlung (Fehlermeldung)

Programmierfehler:

Syntaxfehler: unkorrekte Syntax ➔ leicht zu finden
logische Fehler: z.B. fehlende oder falsche Variableninitialisierung,
falsche Zyklen ➔ schwer zu finden
Laufzeitfehler: Fehler zur Laufzeit, typisch z.B. Bereichsüberschreitung nach mathematischen Operationen, Eingabe einer unzulässigen Zeichenkombination, Division durch Null

Debuggen (entwanzen):

schrittweise Abarbeitung
oder Abarbeitung bis zu Haltepunkten (breakpoints)
Verfolgung von Variablenwerten (Watch-List)

Programmentwicklungszyklus

(compilierende Arbeitsweise)

„Unter Softwaretechnik (engl. Software Engineering) versteht man allgemein die (Ingenieur-) Wissenschaft, die die kosteneffiziente Entwicklung von qualitativ hochwertiger Software behandelt“.

GI (Gesellschaft für Informatik) / Fachgruppe für Softwaretechnik

Phasen des SW-Entwicklungsprozesses (Software Life Cycles)

Herold, Helmut, Lurz, Bruno und Wohlrab, Jürgen. Grundlagen der Informatik. München : Pearson Studium, 2007.

1) Anforderungsanalyse
2) Systementwurf
3) Programmentwurf
4) Implementierung
5) Test (Validierung und Verifikation)
6) Betrieb und Wartung