Betriebssystem SINIX

CES Buch 1

Werkzeuge zur C-Programmierung
Grundlagen und Kommandos

Ausgabe Februar 1986 (CES V1.0B, 1.0C, 2.0)

Bestell-Nr. U2580-J-Z95-1
Printed in the Federal Republic of Germany
6200 AG 7863.(7800)

Siemens Aktiengesellschaft

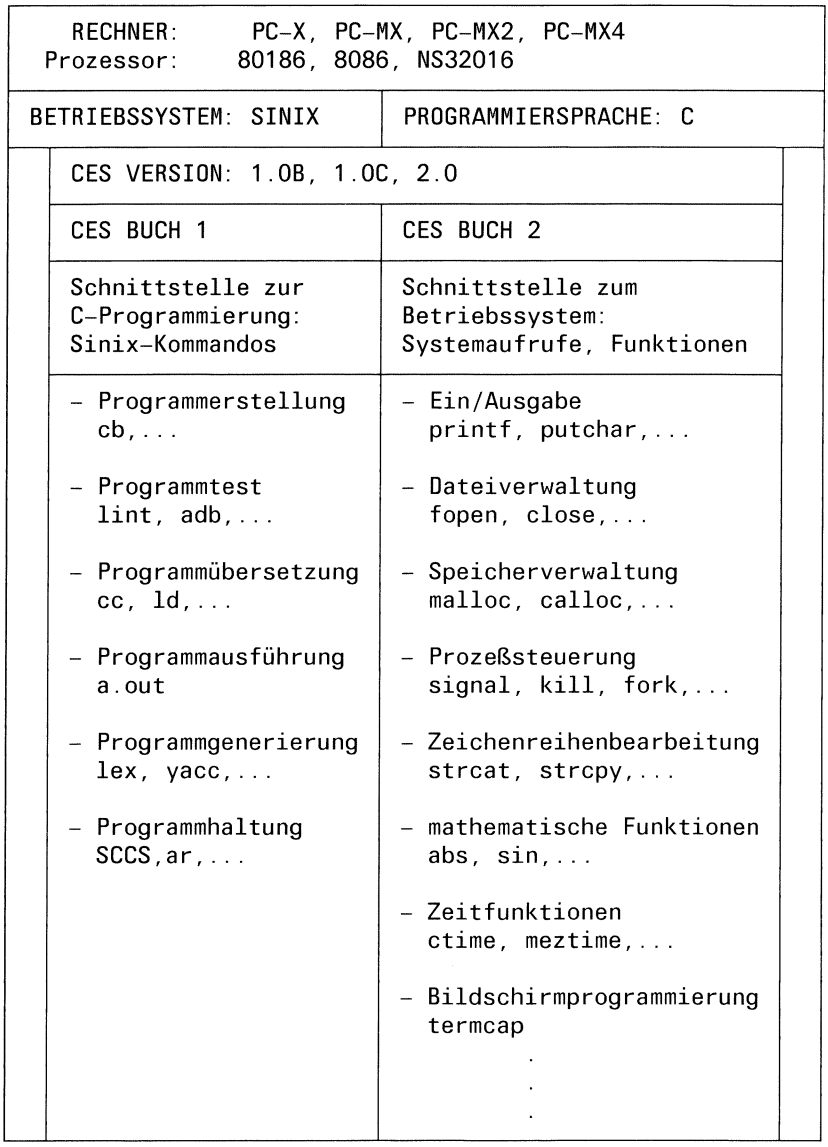


Bild 1-1 Manualübersicht CES-Buch 1 und CES-Buch 2

Vorwort

Was ist CES?

CES (C-Entwicklungssystem) ist ein Software-Paket, das zur C-Programmierung auf einem PC mit Betriebssystem SINIX benötigt wird. Es enthält zwei Klassen von Werkzeugen:

1) Kommandos, die auf Shell-Ebene (d.h. SINIX-Betriebssystem) aufgerufen werden.
   Die Kommandos ermöglichen Ihnen, Ihre fertigen C-Programme auf dem Rechner zu installieren, auszuführen, zu testen und zu verwalten. In Bild 1-1 sehen Sie eine grobe thematische Einteilung der Kommandos, die Ihnen unter CES zur Verfügung stehen.

2) Systemaufrufe, C-Funktionen und Makros, die in einem C-Anwenderprogramm aufgerufen werden.
   Der Programmiersprache C fehlen einige Möglichkeiten, die in anderen Programmiersprachen eingebaut sind. In C gibt es z.B. kein eingebautes Ein/Ausgabe-System, keine dynamische Speicherverwaltung, keine Operatoren für zusammengesetzte Datentypen.
   Ein C-Programmierer, der solche Sachen benötigt, wird allerdings nicht allein gelassen. Ihm stehen Bibliotheken zur Verfügung, in denen er die Funktionen für Aufgaben, wie sie oben genannt wurden, findet.

Wie sieht die CES-Dokumentation aus?

Die Dokumentation des C-Entwicklungssystems besteht aus zwei Manualen:

CES Buch 1 : Grundlagen und Kommandos
CES Buch 2 : Systemaufrufe, C-Funktionen und Makros

Die Aufteilung in zwei Bände ist sinnvoll, weil sie die prinzipielle Einteilung der Werkzeuge in zwei Klassen widerspiegelt (siehe Bild 1-1).

CES Buch 1 beschreibt die Schnittstelle vom Betriebssystem zur C-Programmierung. Es informiert Sie darüber, wie Sie ein C-Programm auf einem SINIX-Rechner installieren und verwalten können.
Kapitel 1 'Grundlagen' beschreibt wesentliche Konzepte des Systems, die Sie vor Anwendung der Werkzeuge kennen müssen. Kapitel 2 'Kommandos' enthält eine ausführliche Beschreibung aller SINIX-Kommandos, die Ihnen mit CES zur Verfügung stehen.
In Kapitel 3 'SCCS' sind die Kommandos des Source Code Control System (SCCS) beschrieben. SCCS ist ein Programmpaket, das die Entwicklung und Verwaltung von Textdateien unterstützt. Es wird eingesetzt bei der Entwicklung großer Programmsysteme und bei der Dokumentationserstellung.

CES Buch 2 ist ein Nachschlagewerk über alle Bibliotheksfunktionen, die mit CES geliefert werden. Es wird bei der Erstellung von C-Programmen benötigt. Sie finden dort eine thematische Zusammenstellung der Funktionen, die Ihnen beim Lösen eines Problems die Suche nach der geeigneten Funktion erleichtert. Die einzelnen Funktionsbeschreibungen erklären dann ausführlich das Verhalten der jeweiligen Funktion, wobei auch mögliche Fehlerquellen, Ausnahmefälle und Stolpersteine berücksichtigt wurden. Fast jede Beschreibung ist durch ein kleines Beispiel veranschaulicht.

Welche Voraussetzungen sollten Sie erfüllen?

Wenn Sie mit CES arbeiten wollen, sollten Sie folgende Kenntnisse bereits besitzen:

1) Grundkenntnisse in SINIX
   (Shell, Dateisystem, ..., siehe Betriebssystem SINIX, Buch 1 /2/)

2) Kenntnis der Programmiersprache C
   Die CES-Dokumentation ist **kein** C-Lehrbuch. Es wird vorausgesetzt, daß Sie mit den Sprachelementen vertraut sind und C-Programme erstellen können.
   Ein gutes C-Lehrbuch ist zum Beispiel 'Programmieren in C' (siehe /8/), die deutsche Ausgabe des Buches der UNIX- und C-Erfinder, W.Kernighan und D.M.Ritchie.
   Die CES-Dokumentation enthält auch **keine** Benutzeranleitung für den Anfänger. Sie sollten also bereits wissen, wie Sie mit Hilfe eines Editors (z.B. ced, siehe Betriebssystem SINIX, Buch 1 /2/) ein C-Programm auf dem Rechner erfassen und welche prinzipiellen Schritte erforderlich sind, um ein ablauffähiges Programm zu erzeugen.

Wenn Sie diese Grundkenntnisse erfüllen, werden Sie problemlos mit der CES-Dokumentation arbeiten können. Auch wenn Sie noch ein C-Anfänger sind, werden Sie sich hoffentlich gut zurechtfinden, da jedes Werkzeug (Kommando oder Funktion) durch ein kleines, leicht verständliches Beispiel veranschaulicht ist. Und Konzepte, die erst für die fortgeschrittenen C-Programmierung interessant sind, werden erklärt.

Wenn Sie mit dem Übersetzer-Generator yacc arbeiten möchten, sollten Sie zusätzlich Kenntnisse aus der Theorie des Übersetzerbaus mitbringen. In der Beschreibung von yacc werden bestimmte Verfahren der Syntaxanalyse (LR(k), LALR(k)) erwähnt, aber nicht erklärt. Die Behandlung dieser Themen sprengt leider den Rahmen dieses Manuals.

Eine Bitte an Sie

Keine erklärende Dokumentation kann perfekt sein. Eine Dokumentation lebt. Sie lebt auch von Ihren Anregungen, Ideen und Verbesserungsvorschlägen. Helfen Sie uns, indem Sie uns Ihre Stolpersteine mitteilen, damit wir sie aus dem Weg räumen können.

Manualredaktion K D ST QM2
Otto-Hahn-Ring 6, 8 München 83

Inhalt

Seite

Seite

Seite

1 Grundlagen

Dieses Kapitel erklärt wesentliche Konzepte des SINIX-Systems.
Wir setzen voraus, daß Sie mit den SINIX Konzepten aus Betriebssystem SINIX, Buch 1 (siehe /2/) vertraut sind. Insbesondere sollten Sie das SINIX Dateisystem kennen. Hier sind zusätzliche Konzepte beschrieben, die erst benötigt werden, wenn man selbst programmiert und zwar:

- Prozesse
- Signale
- Pipe
- Ein/Ausgabe
- Bildschirmprogrammierung

1.1 Prozesse

SINIX ist ein Mehr-Benutzer und Mehr-Prozeß-System.
Das bedeutet, das mehrere Benutzer gleichzeitig mit dem System arbeiten können und gleichzeitig mehrere Prozesse ablaufen können.

1.1.1 Was ist ein Prozeß?

Ein Prozeß ist die Ausführung eines Programmes. Er besteht aus dem ablauffähigen Programmcode, den Programmdaten und einer Reihe prozeßspezifischer Verwaltungsdaten, die zur Steuerung des Programmablaufs erforderlich sind. Man nennt diese Daten **Prozeßkenndaten**.
Im Unterschied zum statischen Programm ist ein Prozeß etwas Dynamisches. Er verändert seinen Zustand im Lauf der Zeit, abhängig von der Situation im System und von den Anweisungen des Programmes, das ihn kontrolliert. Ein Programm kann mehrere Prozesse erzeugen und kontrollieren. Außerdem besteht die Möglichkeit, das Programm, das innerhalb eines Prozesses ausgeführt wird, auszutauschen.

Ein Prozeß kann entweder in der Benutzer- oder Systemphase ablaufen. Ein Prozeß befindet sich in der Systemphase, solange er einen Systemaufruf ausführt. Die Funktion times (siehe CES Buch 2: times) liefert die Zeit, die ein Prozeß benötigt hat, aufgeteilt in die Zeit in der Benutzerphase und die Zeit in der Systemphase.

1.1.2 Prozeßkenndaten

Die Prozeßkenndaten enthalten Information, die zur Verwaltung und Steuerung eines Prozesses erforderlich ist. Die Prozeßkenndaten werden größtenteils in Tabellen geführt (siehe 1.1.3 und Bild 1-2).
Diese Tabellen können nur vom System verwaltet werden.
Allerdings gibt es spezielle Systemaufrufe, die dem Benutzer ermöglichen, viele der Prozeßkenndaten in seinem Programm zu verwenden und sogar zu verändern. Somit ist der Zugriff möglich, aber nicht direkt in die Tabellen, sondern nur über die entsprechenden Systemaufrufe
(siehe CES Buch 2).
Einige Prozeßkenndaten sind nur für das System zugänglich, weil sie zentral verwaltet werden müssen.
Die Gesamtheit der Prozeßkenndaten, die zu einem Prozeß gehören, wird auch **Prozeßumgebung** genannt.

Hier ist eine Liste wichtiger Prozeßkenndaten, die im Programm verwendet werden können:

- Prozeßnummer
- Vaterprozeßnummer
- kontrollierende Datensichtstation
- Prozeßgruppe
- Prozeßgruppennummer
- Prozeßpriorität
- reale Benutzer- und Gruppennummer
- effektive Benutzer- und Gruppennummer
- aktuelles Dateiverzeichnis
- Root-Dateivereichnis
- maximale Dateigröße
- offene Dateien
- Prozeßmaske
- Prozeßzeiten
- Restzeit in der Alarmuhr
- Signalbehandlungen
- Umgebungsvariablen (HOME, PATH, TERM)

Prozeßkenndaten, die ausschließlich vom System verwaltet werden, sind z.B.:

- Prozeßzustand
- zugewiesener Speicherplatz
- Lage des Prozesses (physikalische oder virtuelle Adresse)
- aktuelle Prozeßpriorität

Die Prozeßkenndaten, die für das allgemeine Verständnis eines SINIX Prozesses wichtig sind, werden nachfolgend erklärt:

Prozeßnummer

Jeder Prozeß hat eine vom System zugeordnete Prozeßnummer, unter der er intern geführt wird. Die Prozeßnummer ist eine systemweit eindeutige positive ganze Zahl. Die Prozeßnummer 0 ist für den speziellen Systemprozeß "swapper" (Auslagerer) reserviert. Dieser Prozeß wird bei Systemstart erzeugt und bleibt danach ständig aktiv.
Die Prozeßnummer 1 ist für den speziellen Systemprozeß "init" (Initialisierung) reserviert.
Sie können die Prozeßnummer mit getpid abfragen (siehe CES Buch 2: getpid) aber nicht verändern. Die Zuordnung der Prozeßnummern regelt ausschließlich das System.

Vaterprozeßnummer

SINIX hat eine hierachische, baumartige Prozeßstruktur (siehe 1.1.9).
Jeder Prozeß außer dem Wurzelprozeß (init) hat einen Vater (der Prozeß, der ihn erzeugt hat).
Die Vaterprozeßnummer gehört zu den Kenndaten eines Prozesses.
Ab Version 1.0C können Sie mit getppid die Vaterprozeßnummer abfragen (siehe CES Buch 2: getppid).

Kontrollierende Datensichstation

Die Datensichtstation, die ein Prozeß als erste zum Lesen oder Schreiben öffnet, ist seine kontrollierende Datensichtstation. Nur über diese Station können die Signale 'Abbruch von Datensichtstation' (SIGINT) und 'Unterbrechung von Datensichtstation' (SIGQUIT) an den Prozeß geschickt werden. Ein Programm kann die kontrollierende Datensichtstation über die Pseudogerätedatei /dev/tty ansprechen.

Prozeßgruppe

Jeder Prozeß gehört zu einer Prozeßgruppe. Mitglieder einer Gruppe haben alle dieselbe **Prozeßgruppennummer**. Sie ist die Prozeßnummer des Gruppenchefs. Gruppenchef ist der Prozeß, der die Gruppe gebildet hat. Ein Prozeß kann eine neue Gruppe bilden wenn er die kontrollierende Datensichtstation eröffnet oder einen setpgrp Aufruf durchführt (siehe CES Buch 2: setpgrp). Ein setpgrp Aufruf hebt vorangegangene Gruppenzugehörigkeit auf.
Alle vom Gruppenchef erzeugten Sohnprozesse gehören zu seiner Prozeßgruppe, sofern sie keine neue Gruppe mittels setpgrp bilden. Prozeßgruppen sind im Zusammenhang mit Signalen von Bedeutung. Es gibt die Möglichkeit, Signale an jeden Prozeß aus einer Gruppe zu schicken (siehe 1.2 und CES Buch 2: kill, signal).

Prozeßpriorität

Da in der Regel mehrere Prozesse um die Zuteilung der CPU konkurrieren, gibt es im System einen Steuerungs-(scheduling-)Algorithmus, der den jeweils nächsten Prozeß auswählt. In SINIX erfolgt die Steuerung nach Priorität:
Dem Prozeß mit der höchsten Priorität wird als nächstes die CPU zugeteilt. Um Ungerechtigkeiten bei der Prioritätsvergabe zu verhindern, wird die Priorität eines Prozesses in festen Zeitintervallen neu berechnet. In diese Berechnung gehen die im letzten Zeitintervall verbrauchte CPU-Zeit, die Größe des Prozesses und seine Wartezeit ein.
Prozesse in der Systemphase haben eine höhere Priorität als Prozesse in der Benutzerphase.

Das ps Kommando zeigt zwei Prioritäten an:

- die **aktuelle Priorität**, die der Prozeß gerade besitzt und die bei der nächsten CPU-Vergabe neu berechnet wird
- die **nice Priorität**, die dem Prozeß als Grundpriorität beim Start mitgegeben wurde (siehe CES Buch 2: nice, exec, fork). Sie wird als Steigerungswert in der jeweiligen Prioritätsberechnung berücksichtigt.

Die Priorität, die Sie einem Prozeß mitgeben können, ist eine positive ganze Zahl aus [0, 39]. Dabei bedeutet eine niedrige Zahl hohe Priorität, eine hohe Zahl niedrige Priorität. Nur der Systemverwalter kann die Priorität eines Prozesses erhöhen.

Prozeßzustand

Ein Prozeß kann sich in verschiedenen Zuständen befinden. Die drei wesentlichen Grundzustände sind:

- rechnend (running), wenn er gerade die CPU hat
- verdrängt (suspended), wenn er rechenwillig ist und die CPU nicht besitzt
- wartend (waiting), wenn er auf ein Ereignis (z.B. Tod eines Sohnes) wartet

Darüberhinaus gibt es mehrere Zwischenzustände.

Ein rechnender Prozeß verliert die CPU, wenn er sie entweder freiwillig abgibt, um auf ein Ereignis zu warten (z.B. Ein/Ausgabe-Beendigung) oder ein Ereignis eintrifft, auf das ein Prozeß mit höherer Priorität gewartet hat. Im letzten Fall wird der rechnende Prozeß dann verdrängt.

Benutzer- und Gruppennummern eines Prozesses

Jedem Prozeß werden zwei Arten von Benutzer- und Gruppennummer zugeteilt:

- reale Benutzer- und Gruppennummmer
- effektive Benutzer- und Gruppennummer

Die realen Nummern benutzt das System, um den Prozeß einem Benutzer oder einer Benutzergruppe zuzuordnen. Die realen Nummern sind die Benutzer- und Gruppenkennung des Programmaufrufers.
Die effektiven Nummern bestimmen die Zugriffsrechte eines Prozesses. Sie stimmen mit den realen Nummern des Prozesses überein außer, wenn im Indexeintrag seiner Programmdatei (oder der eines Vorfahren) das s-Bit für Eigentümer bzw. Gruppe gesetzt ist. Dann wird die Eigentümer- bzw. Gruppenkennung dieser Programmdatei zur effektiven Benutzer- bzw. Gruppennummer. Damit ändert der Prozeß seine Identität in Bezug auf die Zugriffsrechte:
Für den Zugriff auf eine Datei sind in diesem Fall nur die Rechte des Eigentümers der Programmdatei von Bedeutung, der Programmaufrufer hat nichts mehr zu melden (siehe CES Buch 2, SINIX Buch 1). Mit dem s-Bit Mechanismus wird in SINIX der kontrollierte Zugriff auf Dateien realisiert.

1.1.3 Interne Verwaltung der Prozesse

Bei der internen Darstellung eines Prozesses unterscheidet man zwei Bereiche:

- den Systembereich (immer im Hauptspeicher)
- den prozeßlokalen Bereich (kann zwischenzeitlich ausgelagert werden)

Im Systembereich steht die System-Prozeß-Tabelle. Sie muß immer vorhanden sein, weil sie die Grundlage der System-Prozeß-Steuerung ist.
In der Tabelle gibt es für jeden Prozeß im System genau einen Eintrag, der u.a. folgende Information enthält:

- Prozeßzustand
  (rechnend, verdrängt, wartend, ...)
- Lage des Prozesses
  (physikalische Adresse im Hauptspeicher oder Blockadresse im Auslagerungsbereich)
- Prozeßgröße
- aktuelle Prozeßpriorität
- reale Benutzer-und Gruppennummer...

Da jeder Prozeß genau einen Eintrag belegt, kann es höchstens so viele Prozesse geben wie es Einträge in der Systemtabelle gibt.

Der prozeßlokale Bereich enthält eine Prozeßtabelle, ein Textsegment, ein Datensegment und ein Kellersegment (siehe Bild 1-2).
Die Bezeichnung "Segment" steht hier für die logische Unterteilung der Bereiche, nicht für die physikalische Realisierung.
Die prozeß-eigene **Prozeßtabelle** enthält im Unterschied zur System-Prozeß-Tabelle die prozeßspezifische Information, die nur benötigt wird, falls der Prozeß im Zustand rechnend ist. Sie kann daher wie der restliche prozeßlokale Bereich zwischenzeitlich ausgelagert werden.
Wie die System-Prozeß-Tabelle kann sie auch nur vom System verwaltet werden.

Die Prozeßtabelle enthält u.a. folgende Prozeßkenndaten:

- effektive Benutzer- und Gruppennummer
- offene Dateien des Prozesses
- Verweis auf das aktuelle und das Root-Dateiverzeichnis
- Signalbehandlungen

Im **Textsegment** steht der ausführbare Programmcode.
In Version 1.0B und 1.0C sind standardmäßig Text- und Datensegment nicht getrennt. D.h. beide Segmente zusammen dürfen nur 64K belegen und der Bereich für das Textsegment ist nicht schreibgeschützt.
Es gibt allerdings die Möglichkeit, Schreibschutz für das Textsegment explizit anzufordern. Dazu geben Sie im cc Kommando den Schalter -n oder -i an. Er bewirkt, daß Text- und Datensegment getrennt werden und für jedes Segment ein Bereich von 64K zur Verfügung steht. Man nennt das Programm in diesem Fall auch mehrfach benutzbar (sharable) d.h., mehrere Benutzer können gleichzeitig das Programm ausführen. Das wird erreicht, indem der Bereich für das Textsegment schreibgeschützt ist und allen Prozessen, die das Programm benötigen, verfügbar gemacht wird.
Ab Version 2.0 ist die physikalische Speicherverteilung intern völlig anders organisiert. Für den Benutzer ist jedoch lediglich wichtig, daß jetzt Text- und Datensegment immer getrennt sind. Insbesondere ist das Textsegment immer schreibgeschützt und mehrfach benutzbar.

Schließlich gibt es noch die Möglichkeit, ein Programm von der Auslagerung auszunehmen. Das erreichen Sie, wenn Sie im Indexeintrag der entsprechenden Programmdatei das sticky-Bit setzen. Dann bleibt der Programmcode immer verfügbar, auch wenn kein Prozeß gerade damit rechnet und er normalerweise auf Platte ausgelagert werden könnte.
Da dieser Mechanismus die zentrale Platzorganisation im System beeinflußt, darf nur der Systemverwalter das sticky-Bit setzen. Es sollten auch nur solche Programme ausgewählt werden, die unbedingt und sehr häufig verwendet werden.

Im **Datensegment** stehen die Benutzerdaten des Programmes. Dieser Bereich ist nochmals unterteilt in den Bereich der initialisierten Daten und den Bereich der nicht initialisierten Daten (auch "bss-Segment" genannt).

Das **Kellersegment** ist für die Organisation des Laufzeitkellers erforderlich.

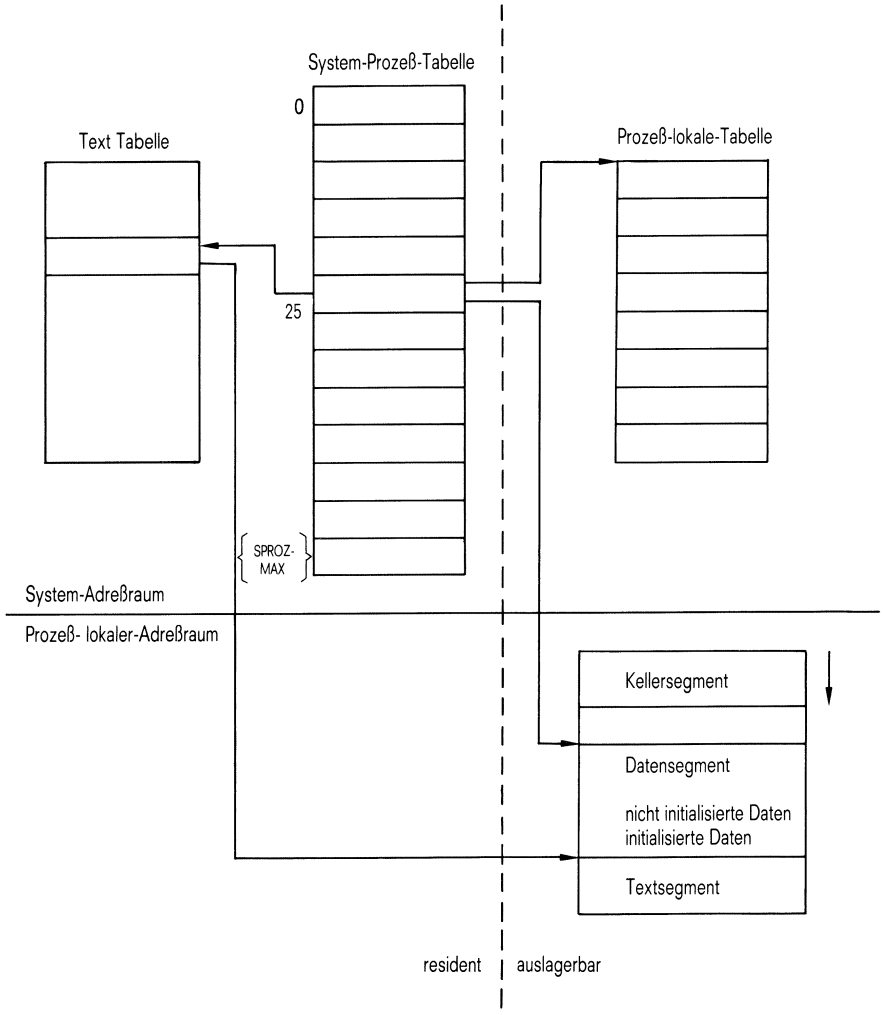


Bild 1-2 logische Speicherverteilung

1.1.4 Beendigung eines Prozesses

Außer der normalen Terminierung oder dem Abbruch eines Prozesses, gibt es die Möglichkeit, einen Prozeß durch den Aufruf der Funktion exit zu beenden (siehe CES Buch 2). Diese Funktion wird häufig eingesetzt, wenn nach einem Fehler abgebrochen werden soll, wie etwa in folgendem Beispiel. Hier soll das Programm abgebrochen werden, falls beim Eröffnen der Datei ein Fehler auftrat:

```
FILE *eingabe;
if ((eingabe = fopen("ein","r")) == NULL)
   {
     printf("Fehler beim Eröffnen der Eingabedatei");
     exit(2);
   }

.
.
```

exit hat einen Parameter, der es ermöglicht, einen definierten Endestatus an den Vaterprozeß zu übergeben.

1.1.5 Programmüberlagerung (exec)

In SINIX gibt es die Möglichkeit, innerhalb eines Prozesses das aktuelle Programm auszutauschen.
Dies geschieht, indem im aktuellen Programm ein exec Aufruf ausgeführt wird (siehe CES Buch 2: exec). Der exec Aufruf bewirkt, daß das aktuelle Programm durch ein neues Programm überlagert wird. Ansonsten bleibt die Prozeßumgebung fast vollständig erhalten.

Zum Beispiel bewirkt der Aufruf

```
.
.
.
execl("/bin/date","datum",NULL);
printf("Datums-Ausgabe klappt nicht!");
```

daß das aufrufende Programm durch das Kommando date überlagert wird. Bei Erfolg wird das Datum in Deutsch ausgegeben und danach der Prozeß beendet. Eine Rückkehr ins aufrufende Programm erfolgt nur, falls der exec Aufruf nicht funktioniert hat. Folglich wird der printf Aufruf in obigem Beispiel nur ausgeführt, falls das date Kommando nicht aufgerufen werden konnte.

Es gibt verschiedene exec Aufrufe. Sie unterscheiden sich nicht in ihrer prinzipiellen Wirkung, sondern nur in Art und Anzahl der Parameter, die übergeben werden können.

1.1.6 Erzeugen von Prozessen (fork)

In SINIX gibt es nur eine Möglichkeit, einen neuen Prozeß zu erzeugen, und zwar mit dem Systemaufruf fork (siehe CES Buch 2: fork).

```
int pnr;
pnr = fork();
```

Die Ausführung von fork bewirkt, daß eine fast vollständige Kopie des aktuellen Prozesses erzeugt wird. Danach existieren zwei beinahe identische Prozesse (sie unterscheiden sich z.B. in ihrer Prozessnummer), die unabhängig voneinander ablaufen.
Der alte Prozeß heißt **Vaterprozeß**, der neue **Sohnprozeß**. Der Sohnprozeß beginnt seine Ausführung mit der Anweisung, die dem fork Aufruf unmittelbar folgt.
An Hand des Ergebnisses von fork kann jeder Prozeß entscheiden, ob er der Vater oder der Sohn ist. Im Vater liefert fork die Prozeßnummer des Sohnes, im Sohn liefert fork das Ergebnis 0.

1.1.7 Kombination von fork und exit

Folgende Kombination von fork und exec bildet den Kern der SINIX Prozeßmechanismen:
Mit fork wird ein neuer Sohnprozeß erzeugt, der dann mittels exec ein anderes Programm aufruft. So entstehen zwei parallel laufende Prozesse, die verschiedene Dinge tun, wie z.B.:

```
int pnr;
char *kommando;

if((pnr = fork()) == 0)
     /* Sohnprozess */
     execl("/bin/sh","sh","-c",kommando,NULL);

else /* Vaterprozess */
.
.
.
```

1.1.8 Prozeßsynchronisation

Um den Ablauf von Vater- und Sohnprozeß aufeinander abzustimmen, hat der Vater die Möglichkeit, auf die Beendigung seines Sohnes zu warten. Dazu muß der Vater den Systemaufruf wait aufrufen (siehe CES Buch 2: wait).
In folgendem Programm z.B. wartet der Vater solange, bis das cat Kommando beendet ist:

```
int pnr;
char *d_name;

if((pnr = fork()) == 0)

   /* Sohnprozess */
   execl("/bin/cat","cat",d_name,NULL);

else if(pnr < 0)
         /* Fehler */
       {
         printf("Fehler bei Prozesserzeugung ");
         exit(2);
       }

     else  /* Vaterprozess */
       {
         while ( wait(&status) != pnr)
         ;
         /* Sohn mit Prozessnummer pnr ist sicher zu Ende */
         /* Auswerten des Endestatus */
         .
         .
         .
```

wait bewirkt, daß der Vater solange angehalten wird, bis der Sohn zu Ende ist. Als Funktionsergebnis liefert wait die Prozeßnummer des Sohnes. Zusätzlich hat wait noch einen Parameter, aus dem der Vater entschlüsseln kann, wie und warum ein Sohn zu Ende ging (normale Terminierung oder Abbruch).

Wenn Sie in Ihrem Programm wie in obigem Beispiel ein Shell-Kommando nur aufrufen wollen und nicht die detaillierte Kontrolle mittels fork-exec-wait benötigen, können Sie die Bibliotheksfunktion system verwenden (siehe CES Buch 2: system). Das obige Beispiel lautet dann einfach:

system("cat d_name");

Im Rumpf der Funktion system wird die richtige Umsetzung in eine fork-exec-wait-Folge automatisch durchgeführt, Sie brauchen sich um nichts mehr zu kümmern.

Weitere Möglichkeiten zur Prozeßsynchronisation und Prozeßkommunikation ergeben sich aus dem Pipe-Mechanismus (siehe 1.3) und den Signalen (siehe 1.2).

1.1.9 Erzeugung der baumartigen Prozeßstruktur

Wenn SINIX neu gestartet wird (booting), wird im letzten Schritt der Urprozeß, Prozeß 0, erzeugt. Prozeß 0 hat aus verschiedenen Gründen eine Sonderrolle unter den Prozessen:

- er kann nicht auf dem normalen Weg erzeugt werden
- er besitzt keinen ausführbaren Programmcode: er besteht lediglich aus einer Datenstruktur, die vom System für die Prozeßverwaltung benötigt wird
- er hat den Status eines Systemprozesses: er ist ausschließlich in der Systemphase aktiv.

Nachdem Prozeß 0 eingerichtet wurde, wird ähnlich wie bei fork durch Anlegen einer Kopie ein neuer Prozeß erzeugt. In diese Kopie wird dann Programmcode eingelagert, der es ermöglicht, das Programm /etc/init aufzurufen. Damit ist die Initialisierung des Systems beendet.
Der init Prozeß ist dafür verantwortlich, die Prozeßstruktur aufzubauen. Er startet die Shell-Prozedur /etc/rc, die einige Verwaltungsarbeiten durchführt (siehe SINIX Buch 1: Systemverwaltung).

Außerdem ist er in der Lage, zumindest zwei unterschiedliche Prozeßstrukturen zu erzeugen:

- Ein-Benutzer-Betrieb
  Im Ein-Benutzer-Betrieb ist lediglich die Konsole angeschlossen und Prozesse, die darauf laufen, erhalten Systemverwalter-Status.

- Mehr-Benutzer-Betrieb
  Im Mehr-Benutzer-Betrieb startet init für jede angeschlossene Datensichtstation einen Initialisierungsprozeß zum Lesen und Schreiben und legt dabei die Standarddateien fest.
  Standardeingabe, -ausgabe und -fehlerausgabe werden auf diese Datensichtstation gelegt und erhalten die Dateikennzahlen:

Standardeingabe	-	Dateikennzahl 0
Standardausgabe	-	Dateikennzahl 1
Standardfehlerausgabe	-	Dateikennzahl 2

  Da bei Programmüberlagerung (exec) und Prozeßerzeugung (fork) der neue Prozeß standardmäßig die offenen Dateien des aufrufenden Prozesses erbt, bleibt diese Zuordnung für alle Folgeprozesse erhalten, wenn sie nicht durch einen Benutzerprozeß explizit aufgehoben wird.

Wenn die Eröffnung der angeschlossenen Datensichtstationen beendet ist, wird für jede Datensichtstation der getty Prozeß gestartet, der darauf wartet, daß ein Benutzer login macht.
Sobald ein Benutzer erfolgreich login gemacht hat, startet i.a. die Shell. Sie ist die Grundlage für die Prozeßhierachie, die der Benutzer während seiner Sitzung aufbaut.

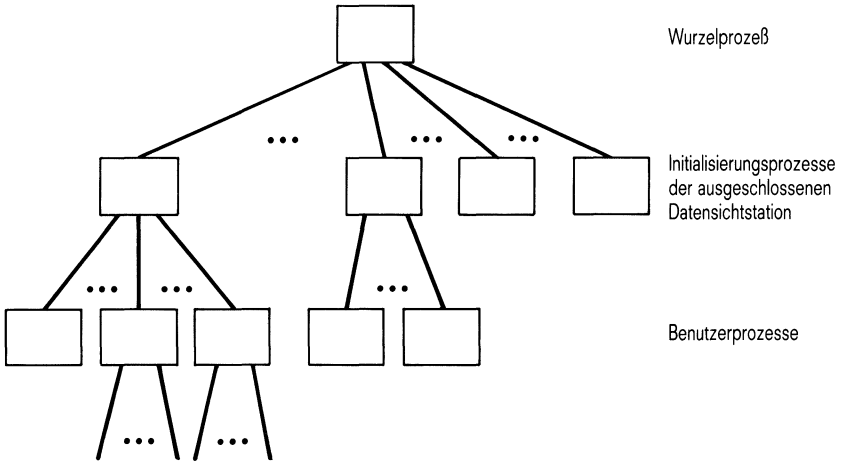


Bild 1-3 Hierachische Prozeßstruktur

Sobald ein Benutzer seine Sitzung und damit seine Ursprungs-Shell beendet, tritt init wieder in Aktion:
Er wartet während der gesamten Lebensdauer des Systems bis einer seiner Sohnprozesse stirbt. Sobald ein Sohn stirbt, erzeugt er einen neuen getty Prozess für die entsprechende Datensichtstation. Somit ist init auch dafür verantwortlich, die Prozeßstruktur im System aufrecht zu erhalten.

1.2 Signale

Signale sind asynchrone Ereignisse, die eine Unterbrechung hervorrufen. Signale, die ein Prozeß empfängt, können auf verschiedene Arten erzeugt werden:

- von außen : durch den Benutzer an der Datensichtstation, der eine Unterbrechungs-Taste ([DEL], QUIT,...) drückt
- von außen : durch einen Prozeß, der ein Signal schickt (siehe CES Buch 2: kill, alarm)
- von innen : durch Programmfehler (Adreßfehler, Anstoß eines ungültigen Befehls, Division durch Null,...)

Anzahl und Art der Signale sind maschinen- und versionsabhängig (siehe CES Buch 2: Anhang). Es gibt allerdings einen Satz von Signalen, die immer unterstützt werden. Wenn Sie in Ihrem Programm nur diese Signale verwenden, bleibt es portabel.

Hat ein Prozeß nichts in Bezug auf ein Signal vereinbart, so wird bei Eintreffen des Signals der Prozeß abgebrochen und der Vater des Prozesses erfährt vom Prozeßende.
Ein Prozeß hat aber auch die Möglichkeit, auf ein Signal zu reagieren. Dazu gibt es den Systemaufruf signal (siehe CES Buch 2: signal, kill). signal hat zwei Parameter, einen für die Signalnummer des Signals und einen für die Aktion, die bei Eintreffen des Signals ausgeführt werden soll. Bei der Angabe der Aktion hat man folgende Möglichkeiten:

- Voreinstellung SIG _ DFL : Prozeßabbruch
- Vordefinierte Funktion SIG _ IGN : Signal ignorieren
- Signalbehandlung gemäß einer vom Benutzer angegeben Funktion:
  Bei Eintreffen des Signals wird der aufrufende Prozeß unterbrochen und die Signalbehandlung ausgeführt. Nach Beendigung der Signalbehandlung wird der Prozeß an der Stelle fortgesetzt, an der er unterbrochen wurde, wenn nicht in der Signalbehandlung exit aufgerufen wurde.

Der Aufruf

signal(SIGINT,SIG_IGN)

bewirkt z.B., daß das nächste SIGINT Signal ignoriert wird. In den meisten Fällen wird nach einer NICHT-Standard-Signalbehandlung die Standardbehandlung nämlich Prozeßabbruch wieder eingestellt. Wollen Sie dies verhindern, müssen Sie vor Eintreffen des Signals signal erneut mit der gewünschten Signalbehandlung aufrufen.

Es gibt ein Signal (SIGKILL), das nicht abgefangen werden kann.

Mit Hilfe von Systemaufrufen können Signale kontrolliert an Prozesse gesendet und und von einem Prozeß empfangen werden.

Mit dem Systemaufruf kill (siehe CES Buch 2: kill) kann ein Prozeß Signale an sich selbst oder an andere Prozesse schicken. kill hat zwei Parameter, der erste gibt an, wer das Signal empfangen soll, der zweite gibt das Signal an, das verschickt werden soll. Der spezielle Wert 0 für den ersten Parameter besagt, daß jedem Prozeß aus der gleichen Prozeßgruppe das Signal geschickt werden soll.

Der Systemaufruf alarm schickt nach einer festgesetzten Zeitspanne ein Signal an den aufrufenden Prozeß.

Außer signal gibt es auf der Empfängerseite den Systemaufruf pause und die darauf aufbauende Bibliotheksfunktion sleep.
Der Systemaufruf pause hält einen Prozeß solange an, bis ein Signal eintrifft, das von kill oder alarm geschickt wurde.

Außerdem hat ein Vaterprozeß die Möglichkeit, einen Sohnprozeß zu tracen (siehe CES Buch 2: ptrace). Das heißt, er kann seinen Sohn an definierten Stellen (Haltepunkte) anhalten, kann ihm Signale schicken, kann ihn verändert weiterlaufen lassen und beenden.

Ansonsten werden Signale durch Drücken spezieller Tasten an der Datensichtstation oder durch Programmfehler ausgelöst.

1.3 Pipe

Eine Pipe ist ein Einweg-Ein/Ausgabe-Kanal, über den kooperierende Prozesse Daten austauschen können.

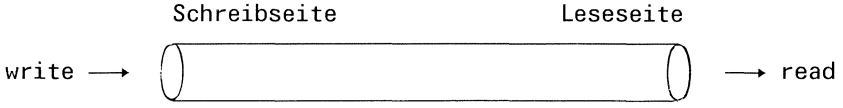


Bild 1-3 Pipe

Statt explizit Dateien zum Austausch der Daten einzurichten, wird beim Pipe-Mechanismus im Hauptspeicher ein Puffer angelegt, über den die Daten wie in einer Warteschlange (FIFO = first in, first out) transportiert werden.
Der Puffer ist systemintern und von außen nicht zugänglich. Der Benutzer kann ihn aber beim Lesen oder Schreiben wie eine Datei ansprechen. Ein Prozeß darf entweder nur auf eine Pipe schreiben oder nur von einer Pipe lesen. Sollen Daten in beiden Richtungen ausgetauscht werden, dann müssen zwei Pipes eingerichtet werden.
Der Systemaufruf zum Einrichten einer Pipe heißt pipe
(siehe CES Buch 2: pipe).
Ähnlich wie die Systemaufrufe open und creat liefert pipe als Verbindung zu dem Puffer Dateikennzahlen und zwar eine für die Leseseite und eine für die Schreibseite. Wie bei einer gewöhnlichen Datei werden diese Dateikennzahlen in späteren elementaren Lese/Schreiboperationen als Dateiargument verwendet.

Eine Pipe für zwei kooperierende Prozesse muß von einem Vater durch einen pipe Aufruf angelegt werden, wie z.B.

```
int pdk[2];
char bereich1[MAX], bereich2[MAX];
int anz;

if(pipe(pdk) != -1)
  if(fork() == 0)

    /* Sohnprozeß */
    /* Schreibseite schließen */
    close(pdk[1]);
    read(pdk[0],bereich2,anz);
    .
    .
  else
    /* Vaterprozess */
    /* Leseseite schließen */
    close(pdk[0]);
    write(pdk[1],bereich1,anz);
    .
    .
    .
```

Die Prozeßbearbeitung wird dabei so gesteuert, daß ein Prozeß, der aus dem noch leeren Puffer lesen will, warten muß, bis Daten in den Puffer geschrieben werden und ein Prozeß, der in den schon zu vollen Puffer schreiben will, warten muß, bis durch einen Lesevorgang Daten aus dem Puffer entfernt wurden.

Der Aufbau einer Einweg-Prozeß-Verbindung vom Vater zum Sohn erfolgt schematisch in folgenden Schritten (siehe Beispiel oben):

1) Der Vaterprozeß erzeugt eine Pipe durch Aufruf von pipe.
2) Der Vaterprozeß erzeugt einen Sohnprozeß mittels fork.
3) Der Vaterprozeß schließt die Leseseite der Pipe.
4) Der Sohnprozeß schließt die Schreibseite der Pipe.

Wenn Sie eine Pipe vom Sohn zum Vater einrichten wollen d.h., der Sohn schreibt auf die Pipe und der Vater liest von der Pipe, müssen Sie lediglich in Schritt 3 und 4 die Rollen von Vater und Sohn vertauschen.

Für gewisse Standardanwendungen dieses Schemas gibt es wie bei der Prozeßerzeugung eine Bibliotheksfunktion, die automatisch die Umsetzung in eine pipe-fork-exec-close-Folge durchführt. Diese Funktion heißt popen.

Zum Beispiel bewirkt der Aufruf

```
FILE *pdz;
char buf[MAX];

pdz = popen("wc","w");
fprintf(pdz,"%s",buf);
```

daß ein neuer Prozeß erzeugt wird, der das wc Kommando ausführt, und eine Pipe eingerichtet wird zwischen dem aufrufenden Prozeß und dem wc Kommando-Prozeß. Die Pipe ist so gerichtet, daß der aufrufende Prozeß auf die Standardeingabe des wc Kommandos schreibt. Durch den nachfolgenden fprintf Aufruf erhält wc als Eingabe also den Inhalt von buf.
popen erzeugt einen neuen Prozeß, der das Kommando ausführt, das Sie als ersten Parameter bei popen angeben und richtet zusätzlich eine Pipe ein zwischen dem aufrufenden Prozeß und der Standardein- oder -ausgabe des aufgerufenen Kommandos. Sie können durch den zweiten Parameter von popen bestimmen, ob der aufrufende Prozeß auf die Standardeingabe des Kommandos schreiben oder von der Standardausgabe des Kommandos lesen soll (siehe CES Buch 2: popen).
Mit dieser Funktion können Sie sich bequem eigene Ein/Ausgabefilter zusammenbasteln.

1.4 Ein/Ausgabe

Wie eingangs erwähnt, gibt es in der Sprache C kein eingebautes Ein/Ausgabe-Konzept. Dieses Kapitel beschreibt, welche Möglichkeiten dem C-Programmierer im CES zur Verfügung stehen, Daten einzulesen und auszugeben.

Wie Sie sicherlich bereits wissen, ist die einheitliche Behandlung verschiedener Ein/Ausgabemedien eine der zentralen Eigenschaften des SINIX Systems (siehe SINIX Buch 1: Dateisystem).
Für den C-Programmierer wirkt sich diese Tatsache so aus, daß er im Programm mit denselben Funktionen z.B. eine Datei auf Platte, den Bildschirm, den Drucker, usw. ansprechen kann. Ein/Ausgabemedium ist immer eine Datei. Der C-Programmierer kann - je nachdem welche Funktion er verwendet - eine Datei auf drei verschiedenen Arten im Programm ansprechen, mit

- dem Dateinamen
  der Dateiname (absolut oder relativ) ist derselbe, den Sie für die Datei auf Shell-Ebene verwenden (siehe SINIX Buch 1: Dateisystem)

- der Dateikennzahl
  sie ist eine positive ganze Zahl, die von Systemaufrufen zur Dateibearbeitung zugewiesen und verwendet wird

- dem Dateizeiger
  er ist ein Zeiger auf eine FILE-Struktur (siehe unten) und wird von den Funktionen aus der Standardein/ausgabe-Bibliothek zugewiesen und verwendet.

Die Aufgaben der Dateibearbeitung können in verschiedene Teilbereiche unterteilt werden, wie Dateizugriff, Dateiverwaltung, Dateiinformation, Ein/Ausgabe, usw. (siehe CES Buch 2: Zusammenstellung der Funktionen). Wir betrachten hier nur eine Auswahl der wichtigsten Funktionen zur Realisierung folgender Aufgaben:

- Verbindungsaufbau zu einer Datei
- Verbindungsabbau
- Zugriffsmethode auf eine Datei
- Eingabe
- Ausgabe

In der C-Standardbibliothek /lib/libc.a gibt es für diese Aufgaben Systemaufrufe und eine Standardein/ausgabe-Bibliothek.

Zu jedem der oben genannten Punkte geben wir jeweils die wichtigsten Systemaufrufe und Funktionen aus der Standardein/ausgabe-Bibliothek an. Die Systemaufrufe sind die elementaren (low level) Ein/Ausgabefunktionen. Die Funktionen und Makros aus der Standardein/ausgabe-Bibliothek beruhen auf den elementaren Systemaufrufen. Sie arbeiten aber zum Teil effizienter, weil sie günstige Pufferungskonzepte automatisch verwenden und sie sind portabel, weil die Standardein/ausgabe-Bibliothek auf jedem UNIX-Rechner mit der gleichen Benuzterschnittstelle verfügbar ist.

1.4.1 Die Standardein/ausgabe-Bibliothek

Wenn Sie in Ihrem Programm Funktionen oder Makros aus der Standardein/ausgabe-Bibliothek verwenden, müssen Sie in den meisten Fällen die include-Datei /usr/include/stdio.h in der Form

#include <stdio.h>

einfügen. Sie enthält Definitionen für oft verwendete Makros (z.B. getchar, putchar) und u.a. folgende vordefinierte Namen:

- EOF Rückgabewert bei Dateiende oder Fehler
- NULL Nullzeiger
- BUFSIZ Standardgröße für den Ein/Ausgabe-Puffer
- FILE Struktur, die Information für gepufferte Ein/Ausgabe enthält

1.4.2 Verbindungsaufbau zu einer Datei

Wenn Sie in Ihrem Programm Daten einlesen bzw. ausgeben wollen, müssen Sie als erstes angeben, auf welcher Datei die Daten stehen bzw. wohin sie geschrieben werden sollen.
Dazu eröffnen Sie in Ihrem Programm eine Verbindung zu der jeweiligen Datei.
Die elementaren Systemaufrufe für diese Aufgabe sind:

- open Eröffnen einer Datei (siehe CES Buch 2: open)
- creat Erstellen oder Eröffnen einer Datei (siehe CES Buch 2:creat)

Ein open Aufruf lautet z.B.:

```
int dk;

dk = open("/usr/sissi/wirrwarr",0);
```

Er öffnet die Datei /usr/sissi/wirrwarr zum Lesen.

Die elementaren Systemaufrufe liefern (bei Erfolg) als Ergebnis die Dateikennzahl zurück, die das System dieser Datei zugeordnet hat. Das System verwaltet intern in der System-Dateien-Tabellen die Dateien nur über ihre Dateikennzahl. Auf der Programmiererseite wird die Dateikennzahl bei der Anwendung von elementaren Dateioperationen gebraucht. Viele dieser Systemaufrufe verlangen die Dateikennzahl als Dateiargument (siehe auch read und write).

In der Standardein/ausgabe-Bibliothek gibt es mehrere Funktionen, die eine Datei eröffnen. Die wichtigste unter ihnen ist fopen.
Obiges Beispiel lautet mit fopen:

```
FILE *dz;
dz = fopen("/usr/sissi/wirrwarr","r");
```

Als Ergebnis liefern die Eröffnungsfunktionen aus der Standardein/ausgabe-Bibliothek den Dateizeiger. Der Dateizeiger zeigt auf die FILE-Struktur, die der Datei vom System automatisch zugeordnet wurde.
Die Definition dieser Struktur steht in <stdio.h>. Sie enthält Information, die spätere Lese/Schreiboperationen aus der Standardein/ausgabe-Bibliothek benötigen, wie z.B. die Lage des Ein/Ausgabe-Puffers, die aktuelle Position des Lese/Schreibzeigers, die Größe der Datei, die Dateikennzahl usw.
Der Dateizeiger wird bei vielen Funktionen aus der Standardein/ausgabe-Bibliothek als Dateiargument verlangt. Er spielt eine ähnliche Rolle wie die Dateikennzahl, ist aber ein höhersprachliches Mittel. Die FILE-Struktur, auf die er zeigt, enthält u.a. die Dateikennzahl.

Was Sie über offene Dateien wissen sollten

Die Anzahl der offenen Dateien pro Prozeß ist beschränkt. Das maximale Limit ist maschinenabhängig (siehe CES Buch 2: Anhang). Auch die Anzahl der offenen Dateien im System ist beschränkt und maschinenabhängig (siehe CES Buch 2: Anhang).

Wenn ein Programm ein neues Programm mittels exec aufruft, übernimmt das neue Programm standardmäßig die offenen Dateien des aufrufenden Programms (siehe 1.1.5 und CES Buch 2: exec). Ab Version 1.0C haben Sie allerdings die Möglichkeit, die Standardeinstellung zu ändern (siehe CES Buch 2: fcntl).

Wenn ein Prozeß einen neuen Prozeß mittels fork erzeugt, haben Vater und Sohn Zugriff auf die offenen Dateien, die zur Zeit des fork-Aufrufes geöffnet sind (siehe 1.1.6 und CES Buch 2: fork). Insbesondere kann der Vater durch dieses Konzept die Dateien für Standardeingabe, -ausgabe und -fehlerausgabe an den Sohn vererben.

Standardverbindungen

In Abschnitt 1.1.9 (Erzeugung der Prozeßstruktur) wurde erwähnt, daß bei Generierung der Prozeßstruktur bereits die Standarddateien zugeordnet werden. Mit der Vererbbarkeit der offenen Dateien ergibt sich daraus, daß für jeden Prozeß automatisch die Dateien für Standardeingabe, -ausgabe und -fehlerausgabe eröffnet sind. Zum Bearbeiten dieser Dateien muß ein Programm also weder open noch fopen aufrufen. Die Standardzuordnungen, die das System vergibt, lauten:

Datei	Dateikennzahl	Dateizeiger
Standardeingabe	0	stdin
Standardausgabe	1	stdout
Standardfehlerausgabe	2	stderr

Alle drei Dateien sind standardmäßig auf den Bildschirm gelegt. Sie haben allerdings die Möglichkeit, diese Zuordnungen in Ihrem Programm neu zu definieren.

1.4.3 Verbindungsabbau

Der Systemaufruf close schließt die Verbindung zu einer Datei, die mittels open eingerichtet wurde, z.B.:.

close(dk); schließt die Datei mit Dateikennzahl dk
close(0); schließt die Standardeingabe

Die Funktion fclose schließt eine Verbindung, die mittels fopen eingerichtet wurde, wie z.B.:

fclose(dz); schließt die Datei mit Dateizeiger dz
fclose(stdin); schließt die Standardeingabe

Bei Prozeßende werden diese Funktionen automatisch aufgerufen. Sie müssen deshalb diese Funktionen nur dann explizit aufrufen, wenn die Gefahr besteht, daß Sie das Limit an offenen Dateien überschreiten.

1.4.4 Zugriffsmethode

Eine Datei in SINIX ist eine Aneinanderreihung von Bytes ohne jegliche Struktur. Folglich ist die Standardzugriffsmethode strikt sequentiell:
Wurde zuletzt das k-te Byte gelesen oder geschrieben, beginnt die nächste Lese/Schreiboperation mit dem (k + 1)-ten Byte.
Für jede offene Datei führt das System einen Lese/Schreibzeiger, der auf das Byte zeigt, das als nächstes gelesen oder geschrieben wird.

Es gibt auch die Möglichkeit, diesen Lese/Schreibzeiger in der Datei nach Belieben zu verschieben und somit einen wahlfreien Zugriff (random access) zu realisieren.
Der Systemaufruf für diese Aufgabe heißt lseek.
Der Aufruf:

```
long pos;
pos = lseek(dk,10L,0)
```

verschiebt den aktuellen Lese/Schreibzeiger in der Datei mit Dateikennzahl dk um 10 Bytes vom Dateianfang. Die neue Position steht in pos.

Die entsprechende Funktion aus der Standardein/ausgabe-Bibliothek heißt fseek.

1.4.5 Eingabe

Die elementare Einleseoperation ist der Systemaufruf read.
Der Aufruf

```
int dk, n, anz;
char puf[MAX];
dk = open("/usr/sissi/kram",0);
n = read(dk,puf,anz);
```

liest anz Bytes aus der Datei mit Dateikennzahl dk in den Bereich, auf den puf zeigt. Sie bestimmen durch Angabe der Byteanzahl, wieviele Bytes bei einem Lesevorgang übertragen werden sollen. Die tatsächliche Anzahl gelesener Bytes kann allerdings von der angeforderten abweichen (siehe CES Buch 2: read)

In der Standardein/ausgabe-Bibliothek gibt es eine Vielzahl von Einlesefunktionen. Sie können von ihrer Wirkungsweise in verschiedene Gruppen eingeteilt werden:

- Funktionen, die von Standardeingabe lesen
  - unformatierte Eingabe
    getchar, getc
  - formatierte Eingabe
    scanf, gets

  Bei diesen Funktionen muß keine Datei als Argument angegeben werden, da die Standardeingabe bereits implizit eingestellt ist.
  Ein Aufruf zum Lesen bis Eingabeende lautet z.B.:

```
int c;
while((c = getchar()) != EOF)
```

- Funktionen, die aus einer beliebigen Datei einlesen

  – unformatierte Eingabe
    fgetc
  – formatierte Eingabe
    fscanf

  Diese Funktionen verwenden den von fopen gelieferten Dateizeiger als Dateiargument.
  Ein Aufruf zum Lesen bis Dateiende lautet z.B.

```
FILE *dz;
dz = fopen("/usr/sissi/gaudi", "r");
while((c = fgetc(dz)) != EOF)
```

Durch die erste Lese/Schreiboperation auf eine Datei, die mittels fopen geöffnet wurde, wird der Datei automatisch ein Ein/Ausgabe-Puffer zugeordnet. Das System besorgt den Platz dafür und stellt in der entsprechenden FILE-Struktur einen Zeiger auf den zugewiesenen Bereich zur Verfügung. Die Standardgröße für diesen Bereich ist in <stdio.h> durch die symbolische Konstante BUFSIZ definiert. Sie ist maschinenabhängig.
Sie haben die Möglichkeit, die Pufferung umzudefinieren. Dazu gibt es Funktionen (setbuf, setbuffer, setlinebuf), mit denen die Pufferung abgeschaltet oder neu gesetzt werden kann.
Diese Funktionen sind alle in CES Buch 2 beschrieben.

1.4.6 Ausgabe

Die elementare Ausgabeoperation ist der Systemaufruf write.
Der Aufruf

```
int dk, n, anz;
char puf[MAX];
dk = open("/usr/sissi/kram",1);
n = write(dk,puf,anz);
```

schreibt anz Bytes aus dem Bereich, auf den puf zeigt, in die Datei mit Dateikennzahl dk. Sie bestimmen durch Angabe der Byteanzahl, wieviele Bytes bei einem Schreibvorgang übertragen werden sollen. Die tatsächliche Anzahl geschriebener Bytes kann allerdings von der angeforderten abweichen (siehe CES Buch 2: write).

In der Standardein/ausgabe-Bibliothek gibt es eine Vielzahl von Ausgabefunktionen. Sie können von ihrer Wirkungsweise in verschiedene Gruppen eingeteilt werden:

- Funktionen, die auf Standardausgabe schreiben

  – unformatierte Ausgabe
    putchar, putc
  – formatierte Ausgabe
    printf

  Bei diesen Funktionen muß keine Datei als Argument angegeben werden, da die Standardausgabe bereits implizit eingestellt ist.
  Ein Programmstück, das die Standardeingabe auf die Standardausgabe kopiert, lautet z.B.:

```
int c;
while((c = getchar()) != EOF)
        putchar(c);
```

- Funktionen, die in eine beliebige Datei schreiben

  – unformatierte Ausgabe
    fputc
  – formatierte Ausgabe
    fprintf

  Diese Funktionen verwenden den von fopen gelieferten Dateizeiger als Dateiargument.
  Ein Programmstück, das die Standardeingabe in die Datei /usr/sissi/gaudi schreibt, lautet z.B.:

```
FILE *dz;
dz = fopen("/usr/sissi/gaudi", "w");
while((c = getchar()) != EOF)
        fputc(c,dz);
```

Was den Ein/Ausgabe-Puffer betrifft, gilt hier das gleiche, was unter dem Abschnitt 1.4.5 (Eingabe) erklärt wurde.

1.4.7 Argumente aus der Aufrufzeile

Der hier beschriebene Mechanismus paßt nicht in das übrige Ein/Ausgabe-Konzept. Es ist eine zusätzliche Möglichkeit, Eingabedaten für ein Programm bereitzustellen.
Wenn ein ablauffähiges Programm aufgerufen wird (von der Shell oder einem anderen Programm), beginnt die Ausführung mit dem Aufruf der entsprechenden main-Funktion. SINIX stellt Argumente, die Sie an der Aufrufstelle angeben, der main-Funktion zur Verfügung. Wenn Sie diese Argumente verwenden wollen, müssen Sie Ihre main-Funktion wie folgt definieren:

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

Die drei Parameter, die der main-Funktion zur Verfügung stehen, werden entweder automatisch bei Programmaufruf oder explizit vom aufrufenden Programm mit aktuellen Werten versorgt.

Dabei bedeutet:

int argc
: Anzahl der Argumente aus der Aufrufzeile
argc hat mindestens den Wert 1, weil der Programmname als erstes Argument zählt.
argc wird automatisch berechnet.

char **argv
: Vektor von Zeigern auf die einzelnen Argumente, mit denen das Programm aufgerufen wurde
Ein Argument ist eine C-Zeichenreihe d.h. ein Vektor von Zeichen, dessen letztes Element das Nullbyte (\0) ist.

char **envp
: Vektor von Zeigern auf die Umgebungsvariablen (HOME, PATH, USER,...).
In vielen Fällen wird vom C-Laufzeitsystem bei Programmstart automatisch ein Zeiger auf den globalen Vektor

extern char ** environ
: übergeben (siehe CES Buch 2: execl, execlp, execv, execvp).

Zur Illustration sehen Sie hier ein Programm, das wie das echo-Kommando seine Eingabe auf die Standardausgabe kopiert:

```
main(argc,argv)
int argc;
char *argv[];

{
  int i;
  for(i=1; i < argc; i++)
     printf("%s%c",argv[i],(i<argc-1)?' ':'\n');
}
```

1.5 Bildschirmprogrammierung

Die angeschlossene Datensichtstation ist ein Gerät, das der C-Programmierer über eine Gerätedatei ansprechen kann. Es zählt zu den zeichenorientierten Geräten. Im Dateiverzeichnis /dev gibt es eine Gerätedatei für jede angeschlossene Datensichtstation. Zur Datenübertragung sind Standardwerte für jede angeschlossene Datensichtstation voreingestellt. Solange ein Programmierer mit diesen Standardwerten zufrieden ist, funktioniert die Datenübertragung von oder auf den Bildschirm wie bei jeder anderen Datei (siehe 1.4 Ein/Ausgabe).
In vielen Fällen (z.B. Erstellen eines Spielprogramms) ist es allerdings erforderlich, daß ein Programmierer bestimmte Eigenschaften des Bildschirms neu einstellen kann. Dazu gibt es im CES Systemaufrufe (ioctl, stty, gtty) und termcap.

Die Systemaufrufe ermöglichen u.a., folgende Eigenschaften umzudefinieren:

- Byteparität
- Übertragungsrate
- Wartezeit nach besonderen Operationen
- Echo ein/aus (Anzeige des getippten Zeichens)
- Einstellen des Verarbeitungs-Modus ...

1.5.1 Verarbeitungs-Modi

Es gibt in SINIX drei Arten der zeichenweisen Verarbeitung.
Der maschinennächste Modus heißt **raw mode** (roher Modus).
In diesem Modus muß das Benutzerprogramm in der Lage sein, jedes Zeichen zu verarbeiten. Denn der Gerätetreiber übernimmt keinerlei Aufbereitungsarbeiten, bevor er das Zeichen weitergibt.
Daher kommt die Bezeichnung "roh". Im einzelnen muß sich der Programmierer um folgende Sachen kümmern:

– alle 8 Bits können für die Übertragung verwendet werden (kein Paritätsbit), d.h. das Programm muß, falls erforderlich, das Paritätsbit selbst generieren und abprüfen.

– alle Zeichen mit besonderer Bedeutung, wie Programmabbruch, Rücksetztaste, Ende der Übertragung, usw. müssen vom Programm selbst behandelt werden.

– die Datenfluß-Kontrolle (XON/XOFF) muß vom Programm synchronisiert werden

Für den Programmierer wesentlich bequemer und trotzdem noch sehr variabel ist der **cbreak mode** (cbreak Modus).
Hier erhält das Programm die Eingabe auch sofort nach Tastendruck und zeichenweise, das System übernimmt jedoch die Aufbereitung von Sonderzeichen. Im Einzelnen gilt:

– wie im "raw mode" wird jedes Zeichen (mit Ausnahme einiger Sonderzeichen) direkt nach Tastendruck an das Programm weitergeleitet

– Paritätsprüfung übernimmt das System

– der Gerätetreiber übernimmt die Datenfluß-Kontrolle

– Sondertasten, die Signale auslösen (z.B. DEL) werden unterstützt

– das EOF-Symbol wird ans Programm weitergeleitet

– Korrektur- und Zeilenlösch-Zeichen muß das Programm selbst behandeln (wegen der zeichenweisen Übertragung)

Der bequemste, vom System standardmäßig eingestellte, Modus ist der **cooked mode** (gekochter Modus). Die Bezeichnung "gekocht" soll ausdrükken, daß die Eingabe erst fertig aufbereitet wird, bevor sie ans Programm weitergeleietet wird. Im einzelnen werden vom System folgende Dienste unterstützt:

– Zur Datenübertragung stellt das System einen Puffer bereit, in dem die Zeichen zunächst gepuffert werden, bevor der auf die Eingabe wartende Prozeß aufgeweckt wird.

– Sonderzeichen wie Rücksetztaste, Zeilen-Lösch-Taste werden nicht in den Puffer geschrieben sondern direkt ausgeführt.

– Die Datenfluß-Kontrolle (XON/XOFF) übernimmt der Treiber direkt.

– Einige Tasten (z.B. DEL, END) senden Signale an die von dieser Datensichtstation aus gestarten Prozesse.

– Das EOF-Symbol wird als Endekriterium an das Programm weitergeleitet.

– Das Paritätsbit wird automatisch generiert und nicht an das Programm weitergeleitet.

- Wenn ein Programm auf Eingabe wartet, wird ihm nach Drücken der Return-Taste die gesamte aufbereitete Zeile übergeben.
- Wenn ein Programm Zeichen ausgibt, übernimmt das System die korrekte Umsetzung von Tabulatoren und anderen Sonderzeichen.

1.5.2 Weitere Möglichkeiten

Mit Hilfe der Systemaufrufe gtty und stty (siehe CES Buch 2: gtty, stty) können Sie noch eine Reihe anderer Eigenschaften des Bildschirms abfragen oder neu setzen.
Hier ist ein Programmstück, das das ECHO abstellt:

```
struct sgttyb bilds;
gtty(0,bilds);
bilds->sg_flags &=~ECHO;
stty(0,bilds);
```

1.5.3 Termcap

termcap ist eine Programmierhilfe, mit der bildschirmorientierte Anwendungen (wie Editoren, Spiele, Menusysteme) unabhängig von dem jeweiligen Bildschirm programmiert werden können.
termcap besteht aus einer Datei /etc/termcap, in der Beschreibungen verschiedener Bildschirmtypen abgelegt sind und einer Bibliothek von C-Funktionen, die den Zugriff auf diese Daten und deren Verarbeitung ermöglichen.
termcap ist ausführlich in CES Buch 2 beschrieben.

2 Kommandos zur Programmentwicklung

Beschrieben sind die SINIX-Kommandos des C-Entwicklungssystems, die Sie zur Programmentwicklung verwenden können. Sie stehen im Dateiverzeichnis /bin und im Dateiverzeichnis /usr/bin (SCCS).

2.1 Kommandos richtig eingeben

Darstellung

Alle SINIX-Kommandos sind dargestellt, wie im folgenden Beispiel:

nm[␣-schalter...][␣datei...]

In dieser Darstellung bedeuten:

nm	ist der Kommandoname. Er ist fett gedruckt. Daran sehen Sie, daß Sie die Angabe "nm" schreiben müssen.
␣	steht für ein Leer- oder Tabulatorzeichen, das zwischen verschiedenen Angaben zu schreiben ist. Sie können auch mehrere Leerzeichen schreiben. Das macht keinen Unterschied.
[ ]	Angaben in eckigen Klammern können Sie weglassen. Bei nm müssen Sie z.B. weder einen Schalter noch einen Dateinamen angeben. Natürlich beeinflußt das die Wirkung des Kommandos. Die eckigen Klammern selbst dürfen Sie nicht schreiben.
–	Vor *schalter* müssen Sie einen Bindestrich schreiben.

schalter — kennzeichnet Wahlmöglichkeiten für die Wirkung des Kommandos. Für *schalter* setzen Sie eine aktuelle Angabe ein.
Meist sind verschiedene Buchstaben als Schalterangabe möglich. Welche, das finden Sie jeweils in der Beschreibung des Kommandos.
Diese Angaben können Sie oft, aber nicht immer (z.B. SCCS) kombinieren:

nm -g	ruft nm mit Schalter g auf,
nm -ng	ruft nm mit Schalter n und g auf.

datei — Für *datei* setzen Sie eine aktuelle Angabe ein, hier z.B. den Namen einer Datei oder einer Bibliothek.
Die jeweilige Bedeutung der Angabe entnehmen Sie der Kommandobeschreibung.

... — heißt, daß Sie die davorstehende Angabe mehrmals machen dürfen, z.B. könnten Sie mehrere Schalter angeben:
nm -n -g oder nm -ng
In diesem Beispiel können Sie auch mehrere Dateiangaben machen, z.B. nm -n adam eva

Beachten Sie

- Manchmal sind die Zeichen "[" und "]" keine Metazeichen, sondern müssen angegeben werden. Das ist dann entsprechend beschrieben.
- Bei manchen Kommandos gibt es Schalter mit vorangestelltem "-", ohne "-" oder auch mit "+". Dann ist dieses Vorzeichen bei jedem Schalter beschrieben. Oft dürfen mehrere Schalter mit Vorzeichen nicht kombiniert werden, sondern müssen durch Leerzeichen getrennt sein.
- Gehört zu einem Schalter ein Argument, dann ist die Syntax dafür jeweils eigens beschrieben, z.B. bedeutet -**f**[␣datei], daß Sie z.B. angeben können: "-f" oder "-f ausgabe".
- Wenn Sie bei einem Kommando einen Schalter angeben, der nicht beschrieben ist, bekommen Sie entweder eine Fehlermeldung oder es kann zu undefinierten Ergebnissen kommen. Bei einigen Kommandos werden falsche Schalter einfach ignoriert.

Kurz zusammengefaßt

Symbol	*Bedeutung des Symbols*
Fettdruck	Konstante, die so einzugeben ist.
normaler Druck	Angabe, für die Sie aktuell etwas einsetzen müssen.
[ ]	wahlfreie Angabe. Die Klammern sind nicht zu schreiben.
...	Wiederholung der vorhergehenden Angabe ist möglich.
Sonderzeichen außer "[","]" und "..."	sind Konstanten, also zu übernehmen. Die Anführungszeichen haben zwar Bedeutung für die Shell, werden aber auch im Text verwendet, um Zeichen oder Zeichenfolgen hervorzuheben, wie z.B. oben. Sie sehen jeweils aus dem Zusammenhang, welche Bedeutung die Anführungszeichen haben.

Wie eingeben?

Kommandos schreiben Sie anschließend an das Bereit-Zeichen ($) in eine Zeile. Sie schließen die Eingabe mit der Taste [↵] ab. [↵] ist für die Shell (siehe Betriebssystem SINIX, Buch 1) das Zeichen, alles Vorhergehende als Kommando zu interpretieren.

Mehrere Kommandos in einer Zeile

kommando1; kommando2; kommando3
: Die Kommandos werden nacheinander angestoßen, ohne Verbindung von Ein- und Ausgaben.

Beachten Sie auch die Möglichkeit, Kommandos mit Pipelines zu ketten (siehe Betriebssystem SINIX, Buch 1, Abschnitt 3.3).

Die Zeile ist zu kurz?

Schreiben Sie am Zeilenende einfach weiter, ohne [↵] zu drücken.

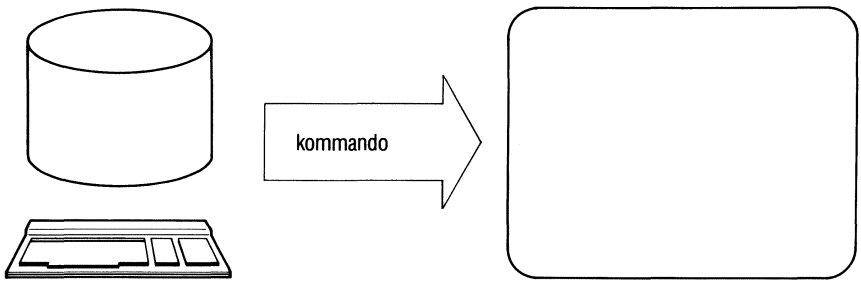
Sie können eine Zeile auch fortsetzen, indem Sie am Ende der Zeile einen Gegenschrägstrich (\) schreiben, [↵] drücken und weiterschreiben.

Daten eingeben

Erwartet ein Kommando Eingaben von der Tastatur, setzt es die Schreibmarke auf den Beginn der nächsten Zeile. Schreiben Sie nun Ihre Eingabe, wobei Sie jede Zeile mit der Taste [↵] abschließen. Das Kommando erhält jede Zeile wie die Zeile einer Datei.
Taste [END] beendet die Dateneingabe, d.h. für das Kommando ist das Dateiende erreicht und das Kommando wird fertig ausgeführt.

2.2 Was Sie zu jedem Kommando wissen sollten

Ein Kommando, das eine Eingabe erwartet, liest diese von der Standard-Eingabe oder von Dateien, deren Namen Sie angeben müssen, wie beim jeweiligen Kommando beschrieben.
Ein Kommando gibt auf die Standard-Ausgabe (Dateikennzahl 1) und auf die Standard-Fehlerausgabe (Dateikennzahl 2) aus.



Standard-Eingabe ist die Tastatur, Standard-Ausgabe und Standard-Fehlerausgabe ist der Bildschirm.

Mit Pipelines und mit Umlenken der Ein- oder Ausgabe schaffen Sie sich mehr Möglichkeiten, z.B.:

- Ein Kommando, das normalerweise von der Tastatur liest, kann auch aus einer Datei lesen oder seine Eingabe von einem vorhergehenden Kommando bekommen.
- Ein Kommando, das normalerweise auf den Bildschirm ausgibt, kann auch in eine Datei ausgeben oder das Ergebnis an ein weiteres Kommando übergeben.

Lesen Sie dazu Betriebssystem SINIX, Buch 1, Abschnitte 3.2 und 3.3.

Gibt ein Kommando auf den Bildschirm aus und ist die Ausgabe länger als eine Bildschirmseite, dann können Sie:

- die Ausgabe anhalten mit [CTRL] [S], die Ausgabe fortsetzen mit [CTRL] [Q],
- die Kommandos "more" bzw. "page" verwenden (siehe Betriebssystem SINIX, Buch 1).

Dateinamen

Wenn Sie Dateinamen oder Namen von Dateiverzeichnissen angeben, haben Sie immer mehrere Möglichkeiten:

- Sie geben einen einfachen Namen an. Damit beziehen Sie sich auf das aktuelle Dateiverzeichnis.
- Sie geben einen Pfadnamen an. Damit können Sie beliebige Dateien und Dateiverzeichnisse im ganzen Dateisystem benutzen, vorausgesetzt, Sie haben die Zugriffsrechte.

Ende-Status

Jedes Kommando liefert einen Ende-Status, der aussagt, wie das Kommando abgelaufen ist. Der Ende-Status steht als Zahlenwert in der Variablen "?". Sie können ihn z.B. mit "echo $?" abfragen.

Ist bei der Beschreibung eines Kommandos weiter nichts angegeben, dann ist der Ende-Status:

0	bei fehlerfreiem Ablauf
3	bei Abbruch des Kommandos mit der Taste [DEL]
ungleich 3	
ungleich 0	bei fehlerhaftem Ablauf

Fehlermeldungen

Zu jedem Kommando gibt es Fehlermeldungen. Sie sind weitgehend selbsterklärend. Fehlermeldungen gehen auf die Standard-Fehlerausgabe. Standard-Fehlerausgabe ist normalerweise der Bildschirm. Sie können sie aber mit "2 > fehler" nach Datei *fehler* umlenken.

Fehlermeldungen des Betriebsystems deuten meist auf einen hardwarebedingten fehlerhaften Ablauf. Als Maßnahme können Sie die Meldung ignorieren und es nochmals versuchen. Wiederholt sich der Fehler mehrmals, verständigen Sie den Systemkundendienst. Solche Fehlermeldungen haben z.B. die Form:

```
ERR ON DEV 1/23
BN=.....
```

2.3 Welches Kommando für welche Aufgabe?

Kommandos sind sehr vielseitig zu verwenden. Die folgende Übersicht teilt die Kommandos nach ihrer hauptsächlichen Funktion ein. Dabei kommen einige Kommandos mehrmals vor.

Dateien verwalten und bearbeiten

Bibliotheken bearbeiten

ar	Bibliotheken verwalten
lorder	Objektmodule ordnen
ranlib	Bibliothek mit einem Inhaltsverzeichnis versehen
tsort	Topologisch sortieren

Dateien verwalten

lorder	Objektmodule ordnen
SCCS-Kommandos	Source Code Control System
tsort	Topologisch sortieren

Editoren unterstützen

ctags	Tags-Datei erstellen

Informationen allgemein

xcho	Tastencode hexadezimal ausgeben

Informationen über Programme und Objektmodule

ctags	Tags-Datei erstellen
lorder	Objektmodule ordnen
nm	Symboltabelle ausgeben
prof	Zeittabelle eines Programms aufstellen
size	Größe einer Objektdatei ausgeben
strings	ASCII-Zeichenreihen in Binärdatei suchen

Programme und Objektmodule erstellen, bearbeiten und verwalten

Ablauffähige Programme erzeugen

adb	C-Programme testen
cb	C-Programme formatieren
cc	Steuerprogramm zum Übersetzen von C-Programmen
ld	Binder aufrufen
lint	C-Programme überprüfen
make	Gruppen von Dateien verwalten
strip	Symboltabellen und Relokationsbits entfernen

Objektmodul-Bibliotheken bearbeiten

ar	Bibliotheken verwalten
cc	Steuerprogramm zum Übersetzen von C-Programmen
ld	Binder aufrufen
lorder	Objektmodule ordnen
ranlib	Bibliothek mit einem Inhaltsverzeichnis versehen
tsort	Topologisch sortieren

Programme generieren

lex	Programme zur Textanalyse generieren
yacc	Parser generieren

Programmsprachen entwickeln und übersetzen

cc	Steuerprogramm zum Übersetzen von C-Programmen
lex	Programme zur Textanalyse generieren
ld	Binder aufrufen
yacc	Parser generieren

Programmveränderungen kontrollieren

SCCS-Kommandos	Source Code Control System

Testhilfen

adb	C-Programme testen
lint	C-Programme überprüfen
nm	Symboltabelle ausgeben
prof	Zeittabelle eines Programms aufstellen

2.4 Vollständige Beschreibung der Kommandos in alphabetischer Reihenfolge

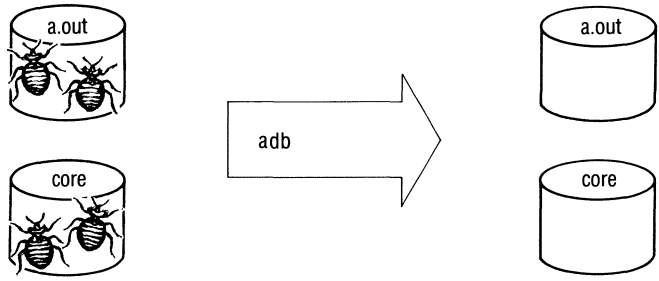
Querverweise zu verwandten Kommandos finden Sie am Ende jeder Kommandobeschreibung, z.B.

> > > > ar verweist auf das genannte Kommando.

Die Beschreibung eines verwandten Kommandos kann auch in "Betriebssystem SINIX, Buch 1 (Kapitel 6)" stehen.

Die SCCS-Kommandos sind im Kapitel 3 beschrieben.

C-Programme testen - a debugger



adb ist eine (interaktive) Testhilfe, die die Fehlersuche in C-Progammen erleichtert. adb ermöglicht es,

- eine Objektdatei, die ein ablauffähiges Programm enthält, und/oder eine Datei, die den Speicherabzug dieses Programms enthält, in verschiedenen Formaten auszugeben, zu analysieren und zu ändern,
- SINIX-Programme, die in der Objektdatei stehen, kontrolliert ablaufen zu lassen,
- allgemein: beliebige Dateien zu analysieren und zu "patchen" (d.h. den Inhalt der Dateien ausbessern).

Sie rufen den adb auf. adb erwartet dann weitere Kommandos von der Standard-Eingabe und antwortet auf der Standard-Ausgabe. adb ignoriert die Taste [DEL]. Er gibt kein Bereit-Zeichen (wie z.B. "$" bei der Shell) aus, wenn er eine Eingabe erwartet. Die adb-Kommandos müssen Sie, wie in der Shell, mit der Taste [↵] abschließen. Mit der Taste [END] können Sie den adb wieder verlassen.

Die folgenden Angaben beziehen sich auf den Prozessor 8086/80186.

Tip

Wenn Ihr Programm fehlerfrei übersetzt wurde, aber nicht fehlerfrei abläuft oder nicht das tut, was es tun sollte, dann können Sie versuchen Ihr Programm mit adb zu testen. Da adb auch vielen SINIX-Kennern große Probleme bereitet und oft Systemkenntnisse erfordert, die dem Normal-Benutzer fehlen, sollten Sie folgendes beachten:

- Noch während der Programmentwicklung lassen Sie Ihr C-Programm von lint prüfen. Erst, wenn lint keine Fehlermeldungen mehr ausgibt oder alle Fehlermeldungen für Sie irrelevant sind, sollten Sie das C-Programm an cc übergeben.
- Um ein ablauffähiges, fehlerhaftes C-Programm zu testen, fügen Sie ins Programm möglichst viele printf-Anweisungen ein. Sie lassen sich z.B. ausgeben, was das Programm gerade gemacht hat oder welche Werte wichtige Variablen angenommen haben. Wenn Sie Ihr Programm dann ablaufen lassen, können Sie die Stellen im Programm erkennen, an denen ein Fehler vorliegt. Wenn Ihr Programm fehlerfrei läuft, entfernen Sie die printf-Anweisungen wieder, die Sie zum Testen eingefügt haben.
- Um die Möglichkeiten von adb kennenzulernen, ist es hilfreich, zuerst die Beispiele durchzuarbeiten.

adb aufrufen

adb[␣**-w**][␣objektdatei[␣speicherabzug]]

-w
: Sie müssen diesen Schalter setzen, wenn Sie in den Dateien *objektdatei* oder *speicherabzug* etwas ändern wollen. Falls eine der Dateien nicht existiert, wird sie erzeugt. Die beiden Dateien werden zum Lesen und Schreiben geöffnet (siehe auch Kommando -w). Normalerweise sind sie nicht zum Schreiben geöffnet.

objektdatei
: In der Datei *objektdatei* steht das ablauffähige Programm, das adb testen soll. Zur leichteren Fehleranalyse sollte die Symboltabelle nicht fehlen.

  Standard (keine Angabe):
  adb untersucht das Programm, das in a.out steht.

Soll adb kein Programm, sondern nur den Speicherabzug untersuchen, so müssen Sie "-" statt eines Dateinamens für *objektdatei* angeben.

speicherabzug

Die Datei *speicherabzug* sollte einen Speicherabzug enthalten, der erzeugt wird, wenn das Programm in *objektdatei* ausgeführt und eventuell fehlerhaft abgebrochen wird. Ein fehlerhaftes Programm muß nicht zwangsläufig zu einem Speicherabzug führen.

Standard (keine Angabe):
adb nimmt den Inhalt der Datei core als Speicherabzug.

Soll adb keinen Speicherabzug untersuchen, so müssen Sie statt eines Dateinamens "-" angeben.

adb verlassen

- entweder durch Drücken der Taste END
- oder durch Drücken von CTRL D
- oder mit einem der adb-Kommandos:
  $q oder $Q

adb-Kommandos eingeben

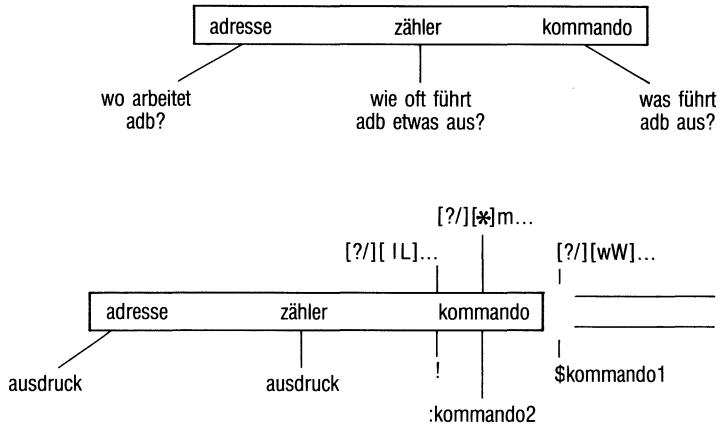
Die allgemeine Form eines Kommandos an adb lautet:

[adresse][,zähler][kommando]

Sie können die Kommandos einzeln eingeben und mit der Taste ↵ abschließen. Wollen Sie mehrere Kommandos auf einmal übergeben, so müssen die Kommandos durch ";" getrennt sein, z.B.

KOMMANDO1;KOMMANDO2;KOMMANDO3

Kommandoübersicht



ausdruck	format	kommando1	kommando2	var	reg
. + ^ # "	b	<datei	b[kom]	0	ax
zahl 'abcd'	o O	>datei	d	1	bx
_symbol symbol	q Q	v	r	2	cx
<var <reg	x X	r	c[sig]	b	dx
	d D	e	s[sig]	d	cs
	u U	m	k	m	ds
(e) *e @e	f F	o		e	ss
-e ~e	c C	d		s	es
	s S	w		t	sp
e1+e2 e1-e2	Y	s			bp
e1*e2 e1%e2	i	c			si
e1&e2 e1|e2	a p	C			di
e1#e2	t	b			ip
	r n	q			fl
	"abc"				
	+ -				
	^				

Adresse, Zähler, Ausdruck

adresse

adb arbeitet an einer aktuellen Arbeitsposition in einer der beiden angegebenen Dateien. Auf die aktuelle Arbeitsposition weist ein Zeiger ".". Ist *adresse* angegeben, so wird der Zeiger "." auf diese Adresse gesetzt.

adresse muß ein *ausdruck* (siehe unten) sein.

Nach dem Start des adb zeigt "." auf die Adresse 0 in der Objektdatei, bzw. (wenn diese nicht untersucht werden soll) im Speicherabzug.

Wie adb eine Adresse interpretiert, hängt vom Kontext ab. Wird z.B. ein Sohnprozeß mit adb untersucht, dann interpretiert adb die Adressen im Adreßraum des Sohnprozesses.

zähler

Bei den meisten Kommandos gibt *zähler* an, wie oft ein Kommando ausgeführt werden soll.

Fehlt der Zähler, so setzt adb ihn auf 1.

Geben Sie als Zähler -1 an, so führt adb das entsprechende Kommando so oft aus, bis irgend ein anderes Abbruchkriterium eintritt, wie z.B. Dateiende. Der Zähler muß ein *ausdruck* sein.

ausdruck

Ein *ausdruck* kann sein:

.	der Wert von ".".
+	der Wert von "." erhöht um das aktuelle Inkrement. Das aktuelle Inkrement ist durch das aktuelle Kommando und das aktuelle Format (siehe unten) festgelegt.
^	der Wert von "." vermindert um das aktuelle Inkrement.
"	die zuletzt angegebene Adresse.
#	die zuletzt angegebene Adresse.
zahl	eine Dezimalzahl, z.B. 256. eine Oktalzahl, z.B. 0543. eine Hexadezimalzahl, z.B. 0xff oder #ff.
'abcd'	eine ASCII-Zeichenkette mit maximal vier Zeichen. """ innerhalb der Zeichenkette muß als "\'" geschrieben werden.

< var	der Wert der Variablen *var* (siehe unten).
< reg	der Wert des Registers *reg* (siehe unten). Die Registerwerte stehen im Kopf des Speicherabzugs.
symbol	der Wert des Symbols *symbol*, der aus der Symboltabelle der Objektdatei genommen wird. Dem Symbolnamen wird "_" (oder "~") vorangestellt, wenn das Symbol extern definiert ist. *symbol* ist eine Zeichenreihe, die aus Groß-, Kleinbuchstaben, "_" und Ziffern bestehen kann und nicht mit einer Ziffer beginnen darf.
_symbol	der Wert des extern definierten Symbols *symbol*. In C wird jedem externen Symbolnamen ein "_" vorangestellt. Wenn interne oder verborgene Variablen mit Namen *symbol* existieren, müssen Sie explizit "_symbol" angeben, um den Wert der externen Variablen zu bekommen.

Es seien *e*, *e1* und *e2* Ausdrücke. Dann ist auch ein *ausdruck*:

(e)	der Wert von *e*.
*e	der Inhalt der mit *e* bezeichneten Adresse in *speicherabzug*.
@e	der Inhalt der mit *e* bezeichneten Adresse in *objektdatei*.
-e	der negative Wert von *e* (als ganze Zahl).
~e	das bit-weise Komplement von *e*.
e1+e2	die Summe von *e1* und *e2* (als ganze Zahlen).
e1-e2	die Differenz von *e1* und *e2* (als ganze Zahlen).
e1*e2	das Produkt von *e1* und *e2* (als ganze Zahlen).
e1%e2	der Quotient von *e1* und *e2* (als ganze Zahlen).
e1&e2	die bit-weise Konjunktion (und-Verknüpfung) von *e1* und *e2*.
e1|e2	die bit-weise Disjunktion (oder-Verknüpfung) von *e1* und *e2*.
e1#e2	*e1* auf das kleinste Vielfache von *e2* aufgerundet, das gleich oder größer als *e1* ist (*e1*, *e2* als ganze Zahlen). Z.B. 10#5 ist 10, 7#3 ist 9.

Register

Der Prozessor 8086 arbeitet mit folgenden Registern:

ax	Akkumulator	sp	Stapelzeiger
bx	Basisregister	bp	Basiszeiger
cx	Zaehlerregister	si	Quellindex
dx	Datenregister	di	Zielindex
cs	Codesegment	ip	Befehlszeiger
ds	Datensegment	fl	Kennzeichenregister
ss	Stapelsegment		(Flags)
es	Extrasegment		

Alle Register haben eine Länge von 16 Bit.

Variablen

adb arbeitet mit einigen internen Variablen, die der Benutzer abfragen und ändern kann.

Die Variablen 0, 1, ..., 9 dienen der Kommunikation mit dem adb. adb belegt

0 mit dem Wert, den adb als letztes ausgegeben hat.

1 mit dem letzten Offset eines Operanden bei einem 8086-Befehl,

- wenn der Befehl im Format i ausgegeben wurde
- und wenn der Operand eine externe Variable ist.

2,...,9 mit dem Wert 0.

Die folgenden Variablen werden nach den Angaben im Kopf des Speicherabzugs, bzw. (wenn kein Speicherabzug existiert) der Objektdatei, zum Teil vorbesetzt. Sie können verändert werden (siehe Beispiele). Die Variablen haben als Inhalt:

b die Basisadresse des Datensegmentes.

d die Größe des Datensegmentes.

m die "magic number" der Objektdatei (0405, 0407, 0410 oder 0411). Die "magic number" ist eine Kennziffer, die angibt, wie Text und Daten geladen werden, z.B. ob sie in getrennten Segmenten stehen.

e	die Startadresse des Programms.
s	die Größe des Stacksegmentes.
t	die Größe des Textsegmentes.

Die eigentlichen Kommandoangaben

kommando — *kommando* gibt an, was adb ausführen soll. Im Kommando können Sie u.a. die aktuelle Datei (*speicherabzug, objektdatei*) festlegen. Sie gilt solange, bis Sie eine neue angeben. Auch das Format, das adb einmal eingestellt hat, gilt bis ein anderes eingeführt wird.
Beim Aufruf des adb ist die Objektdatei die aktuelle Datei und i ist das aktuelle Format (siehe unten).

Bei den Kommandos gibt es folgende Klassen:

?[format...]
adb gibt den Inhalt der Adresse in der Objektdatei dem aktuellen Format bzw. dem Format *format* (siehe unten) entsprechend aus.
Bei einigen Formatangaben gibt adb die aktuelle Adresse als lokales oder globales Textsymbol aus (z.B. ?a).

/[format...]
adb gibt den Inhalt der Adresse im Speicherabzug dem aktuellen Format bzw. dem Format *format* entsprechend aus.
Bei einigen Formatangaben gibt adb die aktuelle Adresse als lokales oder globales Datensymbol aus (z.B. /a).

=[format...]
adb gibt den Wert der Adresse dem aktuellen Format bzw. dem Format *format* entsprechend aus.
Bei einigen Formatangaben wird die aktuelle Adresse als lokales oder globales absolutes Symbol ausgegeben (z.B. =a).

>name — adb weist der Variablen oder dem Register *name* den Wert der Adresse zu. Ohne Angabe einer Adresse wird der Wert von "." zugewiesen.

! — adb ruft die Shell auf und interpretiert den Rest der Zeile als Kommando an die Shell.

$kommando1

kommando1 (siehe unten) gehört zu einer Klasse von adb-Kommandos, die es Ihnen ermöglichen:

- auf Dateien zuzugreifen,
- Informationen auszugeben,
- Voreinstellungen festzulegen oder
- den adb zu verlassen.

:kommando2

kommando2 (siehe unten) gehört zu einer Klasse von adb-Kommandos, mit denen Sie den Ablauf von Programmen (als Sohnprozesse) kontrollieren und beeinflussen können.

?**l**␣wert[␣maske]
/**l**␣wert[␣maske]
?**L**␣wert[␣maske]
/**L**␣wert[␣maske]

adb soll nach dem Speicherinhalt *wert* suchen.
maske, eine ganze Zahl, ist eine Maske. Fehlt *maske*, so nimmt adb die Maske -1.

Beginnend bei ".", sucht adb in der aktuellen Datei ("?" oder "/") nach dem Inhalt *wert*. Dabei werden nur die Bits, die in *maske* auf 1 gesetzt sind, zum Vergleich herangezogen. Bei der Maske -1 (alle Bits auf 1 gesetzt) werden alle Bits verglichen. Der Vergleich umfaßt 2 Bytes, wenn Sie l angeben, und 4 Bytes, wenn Sie L angeben.

Falls die Suche erfolglos war, sollten Sie noch einmal von einer anderen Adresse aus suchen.

- Wenn Sie die Suche zuerst bei einer geraden (ungeraden) Adresse begonnen haben, sollten Sie die zweite Suche bei einer ungeraden (geraden) Adresse anfangen.
- Könnte der gesuchte Wert in einer Adresse vor "." stehen, sollten Sie die Suche weiter vorne in der Datei beginnen.

Wenn adb die Suche erfolglos beendet, bleibt die aktuelle Arbeitsposition unverändert. Bei erfolgreicher Suche, zeigt "." auf die gefundene Adresse.

?**w**␣wert...
/**w**␣wert...
?**W**␣wert...
/**W**␣wert...

adb schreibt den Wert *wert* in die angegebene Adresse der aktuellen Datei. Bei w werden 2 Bytes, bei W 4 Bytes geschrieben. Sie dürfen keine ungeraden Adressen angeben, wenn Sie in den Adreßraum eines Sohnprozesses schreiben wollen.

?[*]**m**[␣w1[␣w2[␣w3]]][/]
?[*]**m**[␣w1[␣w2[␣w3]]][?]
/[*]**m**[␣w1[␣w2[␣w3]]][?]
/[*]**m**[␣w1[␣w2[␣w3]]][/]

adb soll die Werte *w1*, *w2* und *w3*, an b1, e1 bzw. f1 (siehe "Segmentabbildung") zuweisen. Wenn nur ein Wert gegeben ist, ändert adb nur b1, bei zwei Werten nur b1 und e1.

Das erste Zeichen, "?" oder "/", gibt die Datei an, deren Segmentabbildung adb ändern soll. Folgt dem Zeichen "?" bzw. "/" ein "*", so werden die angegebenen Werte an b2, e2 bzw. f2 zugewiesen.

Schließen Sie die Kommandozeile mit "?" oder "/" ab, dann wird die Datei (entsprechend dem Zeichen nach m) *objektdatei* oder *speicherabzug* in Zukunft mit dem Zeichen, das vor m steht, angesprochen.

Z.B. bei Eingabe von "/m?" greift adb in Zukunft mit dem Präfix "/" nicht mehr auf den Speicherabzug, sondern auf die Objektdatei zu. "/m/" stellt wieder den alten Zustand her.

neue zeile

Wenn das zuletzt eingegebene adb-Kommando die aktuelle Arbeitsposition um ein Inkrement erhöhte, soll es zum aktuellen Inkrement werden. adb wiederholt das Kommando mit dem Zähler 1.

Das Format

format

Ein Format *format* besteht aus einer Dezimalzahl und der eigentlichen Formatangabe, die festlegt, in welcher Form die Ausgabe erfolgen soll.

[zahl][eigentlichesformat]

Die Zahl gibt an, wie oft adb die Formatangabe anwenden soll. Fehlt die Zahl, so nimmt adb 1.

Wenn Sie kein Format angeben, so gilt das aktuelle Format, das zuletzt festgelegt wurde. Nach dem Aufruf verwendet adb das Format i.

Durch das aktuelle Format wird auch das aktuelle Inkrement festgelegt. Wenn adb ein Format abarbeitet, dann erhöht sich "." entsprechend dem Inkrement des Formats.

Sie können mehrere Formate direkt nacheinander angeben, ohne Leerzeichen dazwischen.

adb arbeitet mit 32-Bit-Arithmetik.

eigentlichesformat

Die eigentlichen Formatangaben lauten (in Klammern steht die Anzahl der Bytes, um die sich mit dem jeweiligen Format "." erhöht und die das Inkrement festlegen):

b	(1)	1 Byte oktal ausgeben.
o	(2)	2 Bytes (1 Wort) oktal ausgeben (mit vorangestellter 0).
O	(4)	4 Bytes oktal ausgeben.
q	(2)	2 Bytes mit Vorzeichen oktal ausgeben.
Q	(4)	4 Bytes mit Vorzeichen oktal ausgeben.
x	(2)	2 Bytes hexadezimal ausgeben.
X	(4)	4 Bytes hexadezimal ausgeben.
d	(2)	2 Bytes dezimal ausgeben.
D	(4)	4 Bytes dezimal ausgeben.
u	(2)	2 Bytes dezimal ohne Vorzeichen ausgeben.
U	(4)	4 Bytes dezimal ohne Vorzeichen ausgeben.
f	(4)	4 Bytes als Gleitpunktzahl ausgeben.
F	(8)	8 Bytes (4 Worte) als doppelte Gleitpunktzahl ausgeben.
c	(1)	1 Zeichen ausgeben. *adresse* legt fest, welches Zeichen gemeint ist.

C (1) 1 Zeichen ausgeben. adb gibt ein nichtabdruckbares Zeichen, das einen Wert von 000 bis 040 hat, aus als das entsprechende abdruckbare Zeichen mit einem Wert von 0140 bis 0177 und dem Zeichen "@" davor. Das Zeichen "@" wird als"@@"ausgegeben. *adresse* legt fest, welches Zeichen adb ausgeben soll.
adb druckt z.B. für [CTRL] [G] (Wert 07) die Zeichen "@g" (Wert 0147) aus.

s (n) eine Zeichenreihe bis zu einem Nullbyte ausgeben. adresse legt fest, bei welchem Zeichen die Zeichenreihe beginnt. n ist die Länge der Zeichenreihe einschließlich dem Nullbyte.

S (n) eine Zeichenreihe bis zu einem Nullbyte ausgeben. Nichtabdruckbare Zeichen werden wie beim Format C ausgegeben. *adresse* legt fest, bei welchem Zeichen die Zeichenreihe beginnt. n ist die Länge der Zeichenreihe einschließlich dem Nullbyte.

Y (4) 4 Bytes als Datum ausgeben (siehe ctime).

i (n) einen 8086-Befehl ausgeben. Der Befehl umfaßt n Bytes. Die Variable 1 wird eventuell belegt. Beim Kommando "=i" druckt adb das Zeichen "?" aus für die Teile eines Befehls, die Adressen bezeichnen, die vor der aktuellen Adresse liegen.

a (0) Wert der aktuellen Arbeitsposition "." als Symbol ausgeben. Die Symbole müssen von einem bestimmten Typ sein, der davon abhängt, welches der Zeichen "/", "?" oder "=" vor dem Format steht. adb erwartet:

- nach "/" ein lokales oder globales Datensymbol,
- nach "?" ein lokales oder globales Textsymbol,
- nach "=" ein lokales oder globales absolutes Symbol.

p	(2)	adressierten Wert als Symbol ausgeben. Dabei gelten dieselben Regeln wie beim Format a.
nt	(0)	Ausgabe ab dem nächsten Tabulatorzeichen ausgeben. Eine ganze Zahl *n* vor dem t gibt an, in welchem Abstand Tabulatoren zu setzen sind. Z.B. werden bei 8t die Tabulatoren im Abstand von 8 Zeichen gesetzt.
r	(0)	1 Leerzeichen ausgeben.
n	(0)	neue Zeile beginnen.
"abc"	(0)	Zeichenreihe *abc* ausgeben.
^		"." um das aktuelle Inkrement vermindern. adb druckt nichts aus.
+		"." um 1 Byte erhöhen. adb druckt nichts aus.
–		"." um 1 Byte vermindern. adb druckt nichts aus.

kommando1, kommando2

kommando1

<datei	adb soll die Kommandos von der Datei *datei* einlesen.
>datei	adb soll auf die Datei *datei* ausgeben. Falls die Datei nicht existiert, wird sie erstellt.
r	adb gibt die allgemeinen Register und ihren Inhalt aus. Der Befehl, auf den der Befehlszähler zeigt, wird auch ausgegeben. Wenn möglich, wird "." auf den Befehlszähler gesetzt.
b	adb druckt die Haltepunkte (Breakpoints), ihre jeweiligen Zähler und die dazugehörigen Befehle aus.
c	adb listet die aufgerufenen C-Funktionen mit Übergabeparametern und Verweis auf die aufrufende Funktion auf (Stack Backtrace).

C — adb listet die aufgerufenen C-Funktionen mit Übergabeparametern und Verweis auf die aufrufende Funktion auf (Stack Backtrace). Die Kommandos $c und $C liefern die gleiche Ausgabe.

adresse bezieht sich auf den aktuellen Stack-Rahmen. Wenn *zähler* angegeben ist, werden die ersten *zähler* Stack-Rahmen ausgedruckt.

e — adb gibt den Namen und die Werte der externen Variablen aus.

w — Der Wert *adresse* legt die neue Zeilenlänge der Ausgabe fest. Ohne dieses Kommando gibt adb pro Zeile 80 Zeichen aus.

s — Der Wert von *adresse* legt die Distanz fest, bis zu der eine Adreßangabe in der Form

_ symbol + offset

erfolgen soll. *symbol* ist ein Funktionsname oder ein globaler Variablenname. *offset* ist die Entfernung der ausgegebenen Adresse zu *symbol*. Ist *offset* größer als *adresse*, wird die Adresse als Absolutwert (oktal) ausgegeben.

o — adb interpretiert alle ganzen Zahlen in der Eingabe als Oktalzahlen.

d — Das Kommando $o wird abgestellt. adb interpretiert die Eingabe wie üblich.

q — Sie verlassen den adb.

v — adb gibt die Werte der Variablen, die ungleich 0 sind, oktal aus.

m — adb druckt die Segmentabbildung aus.

kommando2

b[kom] adb setzt bei *adresse* einen Haltepunkt (Breakpoint).

Der Haltepunkt wird (*zähler*-1)-mal durchlaufen, wenn das Programm (als Sohnprozeß) abläuft. Der Prozeß wird unterbrochen, wenn der Haltepunkt ein weiteres Mal erreicht wird. Ist z.B. *zähler* gleich 1, dann wird der Prozeß unterbrochen, wenn der Haltepunkt das erste Mal erreicht wird.

Jedes Mal, wenn der Haltepunkt erreicht wird, führt adb das adb-Kommando *kom* aus.
Mit *kom* können Sie auch den Sohnprozeß beeinflussen:

- adb ignoriert z.B. *zähler* und unterbricht den Prozeß beim Haltepunkt, wenn *kom* kein Kommando ist, das den Inhalt einer Adresse ausgibt.
- Wenn z.B. *kom* den aktuellen Arbeitszähler "." auf Null setzt, wird der Prozeß unterbrochen, sobald der Haltepunkt erreicht wird.

Haltepunkte dienen dazu, ein Programm kontrolliert ablaufen zu lassen. Das 1. Beispiel beschreibt die Verwendung von Haltepunkten ausführlicher.

d adb entfernt den Haltepunkt, der bei *adresse* gesetzt ist.

r Das Programm in *objektdatei* soll als Sohnprozeß laufen. adb startet das Programm an der Stelle, die Sie als *adresse* angeben. Ohne Adreßangabe wird das Programm von Anfang an durchlaufen.
zähler gibt an, wieviele Haltepunkte das Programm durchlaufen soll, bevor es unterbrochen wird. Ohne Angabe eines Zählers, wird beim ersten Haltepunkt unterbrochen.
Alle Signale sind möglich, wenn der Sohnprozeß gestartet wird und wenn er läuft. Signale, die für einen Sohnprozeß bestimmt sind, wirken auf adb selbst. adb unterbricht den Sohnprozeß an der Stelle, an der das Signal gesendet wird, merkt sich das Signal und steht wieder für Kommandoeingaben zur Verfügung.

Auf der Kommandozeile können Sie auch Argumente an den Sohnprozeß übergeben. Ein Argument, das mit "<" oder ">" beginnt, gibt an, welche Datei das Programm als Standard-Eingabe bzw. Standard-Ausgabe benutzen soll (siehe "Hinweise").

k — Falls gerade ein aktueller Sohnprozeß läuft, so beendet adb den Sohnprozeß.

c[sig] — adb setzt einen unterbrochenen Sohnprozeß fort.

Der Prozeß wird fortgesetzt an der Stelle, an der er gestoppt wurde, oder an der Stelle *adresse*, wenn Sie eine Adresse angeben.

Wenn Sie eine Signalnummer *sig* angeben, wird das Signal *sig* gesendet. Fehlt *sig* und hat sich adb ein Signal gemerkt, das zur Unterbrechung des Sohnprozesses führte, dann wird dieses Signal gesendet.
Ob der Sohnprozeß abgebrochen wird, hängt von seiner Signalbehandlung ab.
Normalerweise wird der Sohnprozeß mit dem gesendeten Signal abgebrochen.
Mit dem Kommando :c0 wird der Sohnprozeß fortgesetzt und nicht abgebrochen, auch wenn sich adb ein Signal gemerkt hat.

Haltepunkte werden wie beim Kommando :r durchlaufen.

s[sig] — Wenn ein Sohnprozeß existiert, wird er wie bei :c fortgesetzt. Es werden *zähler* Einzelschritte ausgeführt.

Falls kein aktueller Sohnprozeß existiert, dann startet adb das Programm in *objektdatei* (wie bei :r) und übergibt den Rest der Zeile als Argument an den Prozeß (wie bei :r). Ein Signal *sig* wird ignoriert. Es werden *zähler* Einzelschritte ausgeführt.

Beispiele für Kommandoangaben

adb-Kommando	Was macht adb?	Was gibt adb z.B. aus? (Diese Angaben sind von dem Programm, das Sie testen, abhängig)
2?d	adb gibt ab der aktuellen Arbeitsposition im Speicherabzug den Inhalt der nächsten zwei Adressen dezimal aus. Wenn kein Speicherabzug existiert, wird nichts ausgegeben.	0: 32491 5099 0: data address not found
.,2?d ..	adb gibt ab der aktuellen Arbeitsposition in der Objektdatei den Inhalt der nächsten zwei Adressen dezimal aus. "." zeigt danach auf Adresse 2.	start: 32491 -277 0x2
04?i	adb setzt "." auf 04 in der Objektdatei und gibt den Befehl aus, der in Adresse 04 steht.	start+0x4: jmp start+0x6
fehl?s	*fehl* ist eine globale Variable vom Typ char *. Im Programm, das Sie testen steht vor main die Zeile: char *fehl = "FEHLER" adb gibt den Inhalt der Adresse, die mit dem Symbol *fehl* bezeichnet wird, als Zeichenreihe aus. Die Adresse in der Objektdatei wird als Textsymbol ausgegeben.	exit1+0x22: FEHLER
011052=d	adb gibt die Oktalzahl 011052 dezimal aus.	4650
!date	adb interpretiert den Rest der Zeile (nach "!") als Shell-Kommando.	Fri Sep 6 17:11:49 MEZ 1985 !
:r	adb startet das Programm in der Objektdatei.	a.out: running dies ist ein Satz process terminated

adb-Kommando	Was macht adb?	Was gibt adb z.B. aus? (Diese Angaben sind von dem Programm, das Sie testen, abhängig)
:c4	Wenn ein Sohnprozeß gestoppt wurde, wird er an der Stoppstelle fortgesetzt und mit dem Signal 4 abgebrochen. Wenn kein Sohnprozeß existiert, führt adb nichts aus.	a.out running illegal instruction - core dumped process terminated no process
$v	adb gibt die Werte der Variablen, die ungleich 0 sind, aus.	variables 0 = 05 d = 0342 m = 0407 t = 010510
main,5?ia	adb gibt ab der Adresse main in der Objektdatei fünf Befehle aus. "." zeigt danach auf main+0x8.	_main: push bp _main+0x1: mov bp,sp _main+0x3: push di _main+0x4: push si _main+0x5: jmp _main+0x2d _main+0x8:
3Ss	Die Distanzangabe wird auf 3 festgesetzt.	
main,5?ia	Die Ausgabe der Adresse erfolgt ab dem Offset 3 oktal.	_main: push bp _main+0x1: mov bp,sp 0271: push di 0272: push si 0273: jmp 0343 0276:

Die Segmentabbildung

adb interpretiert zwei verschiedene Arten von Objektdateien. Je nachdem, wie Sie ld aufgerufen haben, stehen Text (d.h. das Programm) und Daten in verschiedenen Segmenten oder im selben Segment.

Bei Dateien, die mit

```
$ cc
```

übersetzt werden, sind Text und Daten in a.out nicht getrennt.

Mit dem Schalter -n, also

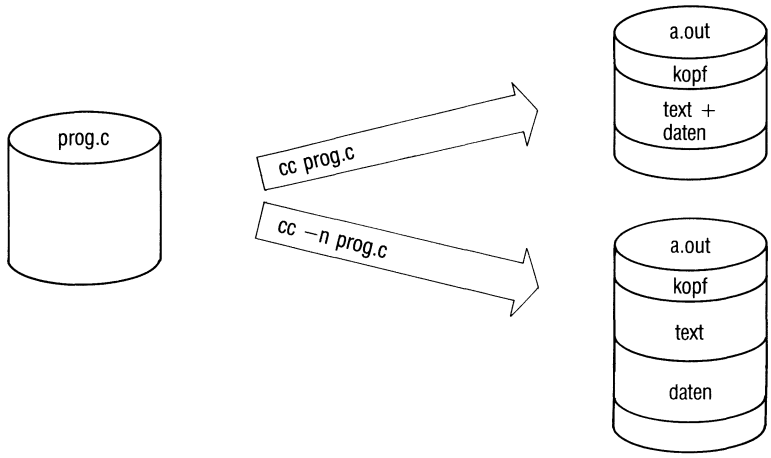
```
$ cc –n
```

erzeugt ld eine Datei a.out, in der Text- und Datenbereich getrennt sind und der Textbereich nicht überschrieben werden kann.

In Standard-Dateien, die keine getrennten Daten- und Textbereiche kennen, ist es für adb unmöglich zwischen Daten und Text zu unterscheiden. Einige symbolische Adressen können dann falsch interpretiert und ausgegeben werden. Beispielsweise kann adb die Adressen von Daten als Offsets von Text-(Programm)symbolen ausdrucken.

"?*" bezeichnet den Datenbereich der Objektdatei, "?" den Textbereich der Objektdatei. Im Datenbereich stehen die Daten vor der Programmausführung. In Objektdateien, in denen Text und Daten getrennt sind, kann z.B. der Datenbereich explizit mit "?*" angesprochen werden.

Im Speicherabzug stehen die Daten wie sie nach einer Programmausführung oder einem Programmabbruch aussehen. Sind in der Objektdatei Text und Daten getrennt, dann enthält der Speicherabzug keinen Programmtext. "/*" bezeichnet entsprechend den Datenbereich des Speicherabzugs, "/" den Textbereich des Speicherabzugs.



adb regelt den Zugriff auf die verschiedenen Segmente in den Dateien durch die **Segmentabbildung**. D.h. es gibt eine Abbildung, die die Adressen, die der Benutzer und adb verwenden, auf die Adressen in der Datei *speicherabzug* oder *objektdatei* (je nachdem, ob "/" oder "?" angegeben ist) abbildet. adb gibt die Segmentabbildung aus mit dem Kommando (siehe 1. Beispiel)

```
$m
```

Wenn adb eine Datei a.out untersucht, in der Text und Daten nicht getrennt sind, und wenn kein Speicherabzug existiert, dann kann die Speicherabbildung z.B. so aussehen:

```
? map           `a.out'
b1 = 0                  e1 = 012526             f1 = 020
b2 = 0                  e2 = 012526             f2 = 020
/ map           `-'
b1 = 0                  e1 = 0100000000         f1 = 0
b2 = 0                  e2 = 0                  f2 = 0
```

Die Segmentabbildung desselben Programms, wenn Daten- und Textsegmente in a.out getrennt sind, lautet:

```
? map           `a.out'
b1 = 0                  e1 = 012200             f1 = 020
b2 = 0                  e2 = 0334               f2 = 012220
/ map           `-'
b1 = 0                  e1 = 0100000000         f1 = 0
b2 = 0                  e2 = 0                  f2 = 0
```

adb benutzt b, e und f um die Segmentabbildung zu spezifizieren. (b1,e1,f1) bezieht sich auf den Text, (b2,e2,f2) auf die Daten. b, e und f sind ganze Zahlen mit 32 Bit. Dabei gilt:

- f1 ist die Länge des Dateikopfes, also der Abstand vom Dateianfang bis zum Textanfang. f2 ist der Abstand vom Dateianfang bis zum Beginn der Daten. Je nachdem, ob Text und Daten getrennt sind, gilt:
  - In Dateien, in denen Text und Daten gemischt sind, sind f1 und f2 identisch.
  - Für getrennte Dateien gilt: f2 = f1 + Länge des Textsegments
    f2 = f1 + (e1 - b1)

- b und e sind die Start- und Endpositionen eines Segments:

b1	Textanfang	e1	Textende
b2	Datenanfang	e2	Datenende

- Aus der Text- bzw. Datenadresse, die der Benutzer und adb verwenden, errechnet sich die Position innerhalb der Datei (hier als "Dateiadresse" bezeichnet) folgendermaßen:

```
Text:
b1 ≤ Textadresse < e1  :      Dateiadresse = (Textadresse -  b1) + f1

Daten:
b2 ≤ Datenadresse < e2  :     Dateiadresse = (Datenadresse - b2) + f2
```

  Textadressen, die größer oder gleich e1 und kleiner als b1 sind, und Datenadressen, die größer oder gleich e2 und kleiner als b2 sind, liegen nicht im Definitionsbereich der Segmentabbildung.

- "?*" bezieht sich auf die Abbildung (b2,e2,f2) der Objektdatei, "/*" auf die Abbildung (b2,e2,f2) des Speicherabzugs.

Anfangs wird die Speicherabbildung von adb vorgegeben. Entspricht eine Datei nicht den Erwartungen von adb oder existiert sie nicht, dann lautet die Speicherabbildung (siehe obiges Beispiel):

`b1 = 0` `e1 = maximale Dateigröße` `f1 = 0`

Sie können die gesamte Datei mit adb überprüfen. Mit den adb-Kommandos

```
?m 100 1000 7
/m 200 1000 80
/*m 1200 2400 1000
```

können Sie die obige Abbildung für getrennte Daten- und Textsegmente ändern zu

```
$m
? map          `a.out'
b1 = 0144              e1 = 01750          f1 = 07
b2 = 0                 e2 = 0334           f2 = 012220
/ map          `-'
b1 = 0310              e1 = 01750          f1 = 0120
b2 = 02260             e2 = 04540          f2 = 01750
```

Hinweis

- Sie können ein Programm in Einzelschritten abarbeiten mit

  : s

  Falls nötig, startet dieses Kommando das Programm und hält nach Ausführung des ersten 8086-Befehls an. Systemaufrufe zählen nicht zu den Befehlen.

- Sie dürfen lokale Variablen nicht als Adresse verwenden.

- Sie müssen Haltepunkte an Programmstellen setzen, auf die bekannte Symbole verweisen. Das sind z.B. alle Funktionsaufrufe. Jede Funktion belegt die ersten Speicherplätze nach dem Funktionsaufruf zum Aufbau eines Stack-Rahmens. Haltepunkte sollten deshalb an die Stelle Funktionsname + 5 gesetzt werden.

- Argumente und Änderungen der Standard-Eingabe oder Standard-Ausgabe werden folgendermaßen ans Programm übergeben:

  :r␣[arg...][< eingabe][> ausgabe]

  Das Programm in *objektdatei* wird neu gestartet mit den Argumenten *arg...*, der Eingabedatei *eingabe* und der Ausgabedatei *ausgabe*. Noch nicht beendete Programmläufe, die adb vorher gestartet hat, werden vor dem Neustart abgebrochen.

- Um eine Adresse zu bezeichnen, kann adb für dieselbe Adresse sowohl ein Text- als auch ein Datensymbol verwenden. Um nicht mit unerwarteten Symbolen konfrontiert zu werden, sollten Sie "?" für Text, d.h. Instruktionen, und "/" für Daten verwenden.

Fehlerdiagnose

- Wenn weder ein aktuelles Kommando noch ein aktuelles Format existiert, wird "adb" ausgegeben.

- adb kommentiert

  – Dateien, die nicht zu öffnen sind,
  – Syntaxfehler,
  – Kommandos, die nicht korrekt beendet wurden.

- Der Ende-Status ist ungleich 0, wenn das letzte Kommando:

  – nicht korrekt ausgeführt werden konnte,
  – einen Ende-Status ungleich 0 meldete.

Version 2.0

- Bei Version 2.0 besteht ein Wort aus 32 Bits, bei den Versionen 1.0B und 1.0C aus 16 Bits. Sie müssen daher bei den Formatangaben die jeweiligen Groß- statt der Kleinbuchstaben verwenden.
- In Version 2.0 sind Text- und Datensegment immer getrennt. Für die Segmentabbildung gilt daher f2 = f1 + (e1 - b1).
- Die Registernamen und die Bedeutung der Register:

```
psr       Flagregister
fsr       Gleitpunktflagregister
sp        Stapelzeiger
fp        Stack-Rahmen-Zeiger

r0  ┐     32-Bit integer        f0  ┐     Gleitpunkt-
...  }    Register               ...  }    register
r7  ┘                           f7  ┘
```

- Beim Disassemblieren werden keine 8086-, sondern 32016-Befehle ausgegeben.

Beispiele

- Die Ausgabe und das Verhalten von adb sind von sehr vielen Faktoren abhängig. Es kann durchaus sein, daß adb bei Ihnen andere Ergebnisse liefert, wenn Sie die folgenden Beispiele durcharbeiten.

1. Pointerfehler erkennen:

   Das Beispielprogramm in der Datei *sosua.c* soll die Zeichenkette "dies ist ein Satz" in "Dies ist ein Satz" umwandeln.

```
char *charzeig = "dies ist ein Satz";
main()
{
        printf("%s\n",charzeig);
        charzeig='D';
        printf("%s\n",charzeig);
}
```

   Das Programm wird folgendermaßen übersetzt:

```
$ cc sosua.c
```

   Der erste Probelauf bringt ein falsches Ergebnis:

```
$ a.out
dies ist ein Satz

$
```

   Das Programm druckt beim zweiten printf-Aufruf offensichtlich nur eine Leerzeile aus. Mit Hilfe des adb suchen wir den Fehler.

   a) *adb aufrufen:*

      Die ausführbare Objektdatei heißt a.out. Ein Speicherabzug existiert nicht, da das Programm regulär beendet wurde. Wir rufen den adb auf:

```
$ adb a.out
```

      Der adb ist nun bereit, interaktiv Befehle entgegenzunehmen. Es erscheint kein Bereitzeichen.

b) *Segmentabbildung ausgeben:*

$m gibt die Segmentabbildung aus, wie oben beschrieben.

```
$m
? map          `a.out'
b1 = 0                  e1 = 011052             f1 = 020
b2 = 0                  e2 = 011052             f2 = 020
/ map          `-'
b1 = 0                  e1 = 0100000000         f1 = 0
b2 = 0                  e2 = 0                  f2 = 0
```

Die Abbildung für den Speicherabzug (/ map) zeigt, daß kein Speicherabzug existiert. Text- und Datenbereich in a.out haben denselben Anfang (b1 = b2) und dasselbe Ende (e1 = e2).

Wie aus e1 und b1 ersichtlich ist, hat das Programm die Länge 011052. Diese Oktalzahl, erkennbar an der führenden Null, kann mit dem adb-Kommando

```
011052=d
                4650
```

in eine Dezimalzahl umgewandelt werden.

Wird beim Übersetzen mit cc oder Binden mit ld der Schalter -n gesetzt, werden Text und Daten getrennt; es ergeben sich verschiedene Werte für e1/e2 bzw. b1/b2.

c) *Breakpoints setzen:*

Die am häufigsten gebrauchte Dienstleistung des adb sind die Haltepunkte (Breakpoints). Es kann vor jedem Maschinenbefehl der ausführbaren Objektdatei ein Haltepunkt gesetzt werden.

Für den Programmierer, der Assembler nicht beherrscht, schränken sich die Möglichkeiten etwas ein. Der C-Compiler generiert keine Symbole für den Code der einzelnen C-Befehle. Darum müssen Sie Haltepunkte an die Programmstellen setzen, auf die bekannte Symbole verweisen. Das sind alle Funktionsaufrufe.

Jede Funktion baut sich einen Stack-Rahmen auf. Dafür werden vier Maschinenbefehle mit insgesamt 5 Byte Länge benötigt. Wir setzen den Haltepunkt deshalb an die Stelle Funktionsname + 5:

```
printf+5:b
```

Mit

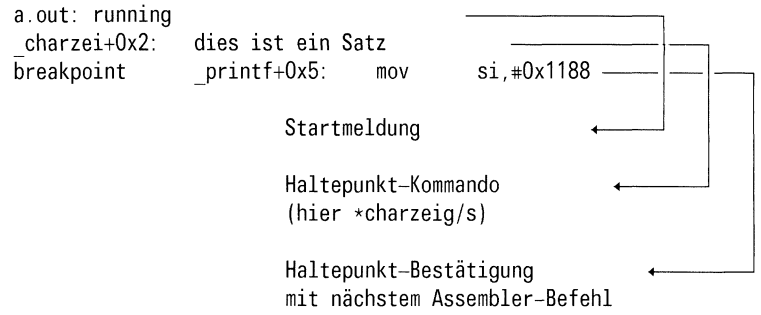
```
$b
breakpoints
count    bkpt              command
1        _printf+0x5
```

lassen wir uns die bis jetzt gesetzten Breakpoints zur Kontrolle ausgeben. Im count-Feld steht der Wert des Zählers, im command-Feld steht das adb-Kommando, das beim Erreichen des Haltepunkts auszuführen ist. Das count-Feld gibt also die Zahl der Durchläufe an, nach denen der Prozeß unterbrochen werden soll (Standardwert 1). adb ignoriert das count-Feld nicht, wenn im command-Feld ein Kommando zur Ausgabe des Inhalts einer Adresse angegeben ist.
Wir wollen beim Aufruf von printf den String ausgeben lassen, auf den *charzeig* zeigt und übergeben stattdessen folgendes Kommando an adb:

```
printf+5:b *charzeig/s
          |  |
          |  |
          |  Kommando, das bei jedem Halt ausgeführt werden soll.
          |  Achtung: Sie können nur global definierte Variable mit einem
          |           Symbol ansprechen.
          |
          |
          hier kann zähler angegeben werden, Voreinstellung ist 1.
          z.B.: printf+5,2:b
```

d) *Programm kontrolliert ablaufen lassen:*

Wir starten das Programm:



Bei Variablen- oder Funktionsnamen sind die ersten sieben Zeichen relevant. adb gibt deshalb statt des vollständigen Symbolnamens *charzeig* nur die ersten sieben Zeichen aus. "_" vor dem Namen zeigt, daß es sich um ein Symbol handelt, das extern im C-Programm definiert wird.

e) *Zur Verwirrung!*

Sie könnten denselben Breakpoint wie oben auch so setzen:

```
printf+5:b *charzeig/s
```

Nach dem Programmstart erfolgt die Ausgabe

```
a.out: running
exit1+0x22:       dies ist ein Satz
breakpoint        _printf+0x5:    mov     si,#0x1188
```

Vor "dies ist ein Satz" steht jetzt die Adresse exit1 + 0x22. Diese Adresse ist dieselbe wie charzei + 0x2. Mit dem Kommando "=x" lassen sich beide Adressen hexadezimal ausgeben.

```
exit1+0x22=
                  0x1154
_charzei+0x2=
                  0x1154
```

exit + 0x22 und charzei + 0x2 bezeichnen dieselbe Adresse.

adb verwendet bei der Angabe von Adressen in der Datei a.out entweder Text- oder Datensymbole. Nach "?" gibt adb die Adresse als Text-, nach "/" als Datensymbol aus. exit1 + 0x22 bezeichnet die Adresse als Teil des Textes, charzei + 0x2 als Teil der Daten. Um nicht von unerwarteten Symbolnamen verwirrt zu werden (wie exit1 + 0x22, wenn Sie Daten anschauen möchten), sollten Sie in einem solchen Fall "?" für Instruktionen und "/" oder "?*" für Daten verwenden.

f) *C-Stack Backtrace ausgeben:*

Über den Stack werden beim Funktionsaufruf Parameter an die Funktionen übergeben und die Rückkehradresse deponiert. Diese Information können Sie mit dem Befehl $c auswerten und in einer lesbaren Form ausgeben.

```
$c
_printf(0x1166, 0x1154) from _main+0xe
_main(0x1, 0xff9e, 0xffa2)      from s1+0x1b
```

g) *Was kann man aus dieser Information herauslesen?*

Das Hauptprogramm main wird von der C-Startfunktion s1 aufgerufen. In den runden Klammern, nach "main", stehen:

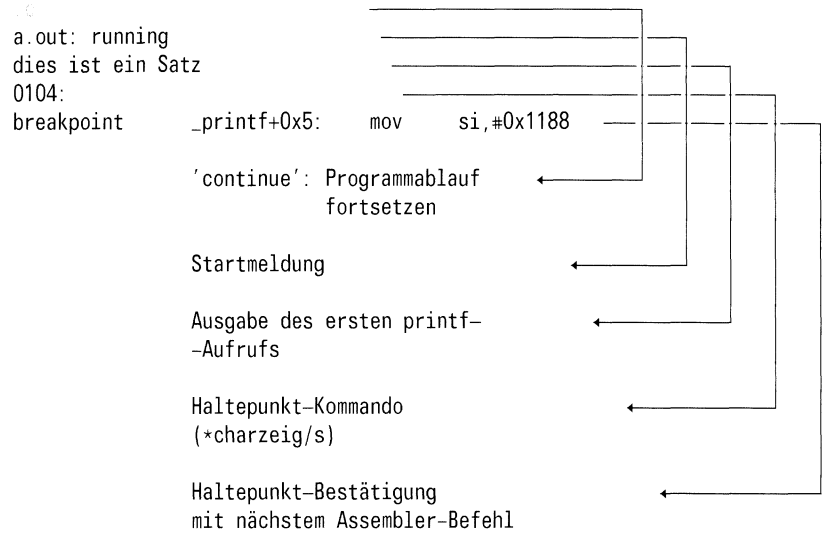
- der Argumentzähler (argc), der den Wert 0x1 hat (dezimal 1),
- der Zeiger auf die Argumenttabelle (argv),
- ein Zeiger auf die Tabelle der Programmumgebung.

```
0177656:        a.out
0177664:        HOME=/usr/andreas
```

printf wird von main aufgerufen. Zwei Argumente, Zeiger auf Character, werden übergeben:

```
_charzei+0x14:  %s
_charzei+0x2:   dies ist ein Satz
```

Offensichtlich werden die Zeichenketten richtig übergeben. Das Ergebnis des weiteren Programmablaufs bestätigt das:



Beim zweiten Aufruf von printf zeigt *charzeig* nicht mehr auf die Zeichenkette, die ausgegeben werden soll (Adresse 0x1154 mit Inhalt "dies ist ein Satz"), sondern auf die Adresse 0104 mit nicht ausdruckbarem Inhalt. Die Adresse 0104 = 0x44 entspricht dem ASCII-Zeichen "D", das wir dem Anfang der Zeichenkette zuweisen wollten (Zeile 5 im Beispielprogramm).

Hier liegt der Fehler!

Der Buchstabe "D" wurde statt in die Zeichenkette selbst, in den Zeiger auf diese Zeichenkette geschrieben. Als Ergebnis zeigt *charzeig* in einen Speicherbereich mit Anfangsadresse 0x44 (= D), in dem nicht "dies ist ein Satz" steht.

Zeile 5 lautet richtig:

```
*charzeig='D';
```

Wir brechen die adb-Sitzung ab und befinden uns danach in der Shell:

```
$
```

2. Dateiinhalt formatiert ausgeben und verändern ("patchen"):

   Am Programm des 1. Beispiels soll gezeigt werden, wie Sie einen Dateibereich ausgeben und verändern können. Sie müssen adb mit dem Schalter w aufrufen, wenn Sie einen Dateiinhalt ändern wollen.

   ```
   $ adb -w
   ```

   a) *Dateiinhalt formatiert ausgeben:*

   Das adb-Kommando

   ```
   _charzei,8?4o4^8Cn
   ```

   führt zu der Ausgabe

```
exit1+0x20:     010524    064544    071545    064440    T@qdies i
                072163    062440    067151    051440    st ein S
                072141    0172      071445    012       atz@`%s@j@`
                071445    012       067050    066165    %s@j@`(nul
                024554    031000    032061    032067    l)@`21474
                031470    032066    070       012054    83648@`,@t
                0         012054    01        0         @`@`,@t@a@`@`@`
                0         0         0402      0         @`@`@`@`@b@a@`@`
```

   Das Kommando gibt an, daß

   - der Inhalt von 8 Speicherplätzen der Objektdatei ("?")
   - ab der Adresse _charzei
   - im Format "4o4^8Cn"

   ausgegeben werden soll.

   _charzei ist die Adresse, 8 der Zähler und "?4o4^8Cn" das eigentliche Kommando.

   adb interpretiert '4o4^8Cn' wie folgt:

4o	Drucke vier Oktalzahlen von zwei Byte (= 1 Wort) Länge.
4^	Gehe viermal um das aktuelle Inkrement zurück. D.h. adb setzt die aktuelle Adresse auf den Anfang der vier Oktalzahlen zurück.
8C	Drucke acht aufeinanderfolgende Zeichen mit folgender Vereinbarung: Jedes Zeichen im Bereich von 0 bis 31 wird gedruckt als Klammeraffe "@" gefolgt von dem entsprechenden Zeichen im Bereich von 96 bis 127. Der Klammeraffe selbst wird als "@@" ausgegeben.
n	Gib das Zeichen "Neue Zeile" aus.

b) *Dateiinhalt verändern:*

Als nächstes wollen wir mit Hilfe des adb den Anfang der Zeichenreihe "dies ist ein Satz" verbessern zu "Dies ist ein Satz", d.h. wir suchen den Anfang des Satzes, "di", und ändern ihn zu "Di".
Die entsprechenden adb-Kommandos lauten:

```
?i "di"
```

adb sucht nach dem Wert 'di' und gibt die Adresse aus, in der der Wert steht:

```
exit1+0x22
```

Dem Kommando

```
?w 'Di'
```

folgt die Ausgabe:

```
exit1+0x22:          064544   =          064504
```

Die formatierte Ausgabe lautet:

```
exit1+0x22,8?4o4^8Cn
exit1+0x22:      064504  071545  064440  072163  Dies ist
                 062440  067151  051440  072141   ein Sat
                 0172    071445  012     071445  z@`%s@j@`%s
                 012     067050  066165  024554  @j@`(null)
                 031000  032061  032067  031470  @`2147483
                 032066  070     012054  0       648@`,@t@`@`
                 012054  01      0       0       ,@t@a@`@`@`@`@`
                 0       0402    0       0       @`@`@b@a@`@`@`@`
```

adb hat die Veränderung richtig ausgeführt. Wir verlassen den adb.

```
$c
$
```

3. Disassemblieren:

   Das adb-Kommando

   ```
   main,7?ia
   ```

   gibt den Inhalt der Objektdatei aus:

```
_main:          push    bp
_main+0x1:      mov     bp,sp
_main+0x3:      push    di
_main+0x4:      push    si
_main+0x5:      jmp     _main+0x2d
_main+0x8:      push    _charzei
_main+0xc:      mov     di,#0x1166
_main+0xf:
```

   Ab dem Symbol main werden 7 Zeilen aus dem Textbereich ("?") disassembliert. Die Erweiterung a bewirkt, daß vor jeder Zeile die Adresse relativ zum letzten Symbol angegeben wird. Die ersten vier Anweisungen dienen, wie im obigen Beispiel beim Setzen der Haltepunkte angedeutet, zum Aufbau des Stack-Rahmens für Funktionen.

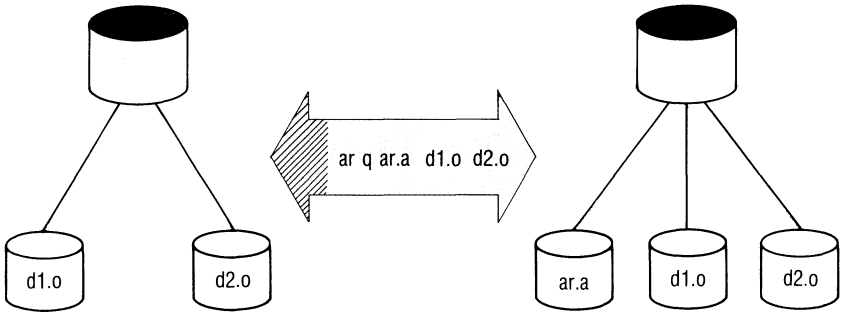
   Auch ohne Kenntnisse von Assembler können die disassemblierten Speicherinhalte zur Orientierung dienen und über Funktionsaufrufe und externe Variablen informieren.

Dateien

a.out	enthält das ausführbare Programm, das adb untersuchen soll, wenn Sie *objektdatei* nicht angeben.
core	enthält den Speicherabzug, den adb untersuchen soll, wenn Sie *speicherabzug* nicht angeben.

> > > > ctime, ptrace, signal

Bibliotheken verwalten - archive and library maintainer



ar erstellt und verwaltet Bibliotheken. Es werden mehrere Dateien in einer einzigen Datei, der Bibliothek, zusammengefaßt.

Die Bestandteile von Bibliotheken sind häufig Objektmodule. Die Module müssen dann beim ld- oder cc-Aufruf nicht einzeln übergeben werden. Es genügt, wenn Sie den Namen der Bibliothek angeben.

ar␣schalter[␣position]␣archiv[␣datei...]

schalter — Ein Schalter besteht aus genau einem Hauptschalter, eventuell verbunden mit einem oder mehreren Zusatzschaltern. Die Reihenfolge der Schalter ist beliebig.

Hauptschalter:

d, r, q, t, p, m, x.

Zusatzschalter:

v, u, a, i, b, c, l.

Ausnahme: u kann auch ohne einen Hauptschalter als Schalter verwendet werden (siehe unter u).

Beispiele für Schalter:

dv, vd, ruv, uv, vur, mbv, tv, vq, x, p.

Hauptschalter:

d *Datei aus der Bibliothek entfernen:*

ar entfernt die unter *datei* genannten Dateien aus der Bibliothek *archiv*. Die Bibliothek muß existieren und die Dateien enthalten. Falls keine Dateien angegeben sind, bleibt die Bibliothek unverändert erhalten. Steht eine Datei gleichen Namens mehrmals in der Bibliothek, entfernt ar nur die erste Datei mit diesem Namen.

r *Datei in der Bibliothek ersetzen:*

Dabei sind folgende Fälle möglich:

- *archiv* existiert und eine Datei *datei* ist bereits in der Bibliothek vorhanden:
  ar ersetzt *datei* in der Bibliothek durch *datei* außerhalb der Bibliothek. Die Dateien bleiben in ihrer ursprünglichen Reihenfolge in der Bibliothek stehen.
- *archiv* existiert und *datei* befindet sich noch nicht in der Bibliothek:
  ar hängt *datei* ans Ende der Bibliothek an.
- *archiv* existiert im aktuellen Dateiverzeichnis noch nicht:
  ar legt die Bibliothek an und faßt die angegebenen Dateien in der Bibliothek zusammen.

Die angegebenen Dateien müssen außerhalb der Bibliothek existieren.

Der Schalter ru veranlaßt dieselben Aktionen wie r, außer daß die Datei in der Bibliothek nur dann ersetzt wird, wenn die außerhalb der Bibliothek stehende Datei neueren Datums ist als die Version, die sich bereits in der Bibliothek befindet.

Bei zusätzlichen Schaltern a, b oder i muß eine Datei aus der Bibliothek als Position angegeben sein. Dateien, die noch nicht in der Bibliothek stehen, werden dann nach (a) oder vor (b oder i) *position* eingefügt. Die Reihenfolge der Dateien, die bereits in der Bibliothek stehen, bleibt unverändert.

q *Datei "schnell" in die Bibliothek kopieren:*

ar hängt die genannten Dateien "schnell" an das Ende der Bibliothek an. Die Schalter a, b oder i sind nicht erlaubt. ar prüft nicht, ob die Dateien bereits in der Bibliothek vorhanden sind oder mehrmals genannt werden. Es können also mehrere Dateien gleichen Namens in der Bibliothek stehen. Eine Bibliothek, die noch nicht existiert, wird angelegt.

Dieser Schalter hilft beim schrittweisen Erstellen von großen Bibliotheken, da der Zeitaufwand für die Suche nach einem Eintrag mit der Anzahl der Dateien quadratisch wächst.

t *Inhaltsverzeichnis der Bibliothek ausgeben:*

Wenn Sie keine Datei angeben, gibt ar das Inhaltsverzeichnis der Bibliothek auf der Standard-Ausgabe aus.

Geben Sie eine oder mehrere Dateien an, listet ar nur die Namen dieser Dateien auf. Die Namen werden in der Reihenfolge ausgegeben, in der die entsprechenden Dateien in der Bibliothek stehen.

Verwenden Sie zusätzlich den Schalter v, werden mit jedem Dateinamen noch ausführliche Informationen zu der jeweiligen Datei ausgegeben. Die Ausgabe ähnelt der Ausgabe des Kommandos ls -l (siehe Betriebssystem SINIX, Buch 1).

p *Dateiinhalt ausgeben:*

ar gibt die Inhalte der angegebenen Dateien auf der Standard-Ausgabe aus. Die Dateien müssen in der Bibliothek vorhanden sein.

Verwenden Sie zusätzlich den Schalter v, dann steht vor dem Inhalt jeder Datei noch der Name der Datei.

m *Datei in der Bibliothek verschieben:*

Die genannten Dateien müssen bereits in der Bibliothek stehen. ar verschiebt sie an das Ende der Bibliothek, wenn kein zusätzlicher Schalter angegeben ist.

Falls Sie einen der Schalter a, b oder i angeben, müssen Sie auch eine Position vorgeben. Die genannten Dateien werden dann nach (a) oder vor (b oder i) *position* eingeschoben.

x *Datei aus der Bibliothek kopieren:*

ar kopiert die angegebenen Dateien aus der Bibliothek in das aktuelle Dateiverzeichnis. Der Name einer kopierten Datei ist derselbe wie der Name der Datei in der Bibliothek. Die Bibliothek bleibt unverändert.

Fehlt *datei*, so werden alle Dateien der Bibliothek kopiert.

Zusatzschalter:

v — Siehe Schalter t und p.

Steht v zusammen mit einem anderen Hauptschalter (d, r, q, m oder x), wird auf der Standard-Ausgabe die Ausführung des ar-Befehls für jedes betroffene Mitglied der Bibliothek bestätigt.

u — Siehe Schalter r.

Bei einem Schalter, der die Buchstaben r und u enthält, kann r weggelassen werden. Der ar-Befehl führt in beiden Fällen dasselbe aus.

a, b, i — Positionsangaben.

Siehe *position* und Schalter r, m und u.

c — c unterdrückt eine Meldung auf der Standard-Fehlerausgabe, wenn die Bibliothek im aktuellen Dateiverzeichnis noch nicht vorhanden ist und erst erstellt werden muß (siehe r, q, u).

l — Beim ar-Befehl legt das System Zwischendateien im Dateiverzeichnis /tmp an. Diese Dateien werden nach Ausführen des ar-Befehls wieder gelöscht. Wenn Sie den Schalter l angeben, schreibt ar die Zwischendateien ins aktuelle Dateiverzeichnis und nicht nach /tmp.

position — Bei den Schaltern a, b oder i, müssen Sie eine Position angeben. *position* muß der Name einer Datei sein, die bereits in der Bibliothek steht. Die Position bezeichnet die Stelle in der Bibliothek, vor (b oder i) oder nach (a) der ar eine Datei einfügen soll.

archiv — *archiv* ist der Name einer Datei, in der (Bibliotheks-) Dateien zu einer Bibliothek zusammengefaßt sind oder zusammengefaßt werden sollen. Die Bibliothek heißt also *archiv*. Bibliotheksnamen enden meistens mit ".a".

datei — *datei* ist ein Dateiname. Sind mehrere Dateien angegeben, so behandelt ar sie in der angegebenen Reihenfolge.

Hinweis

Wenn Sie beim ar-Aufruf einen Dateinamen mehrmals angeben, kann es vorkommen, daß die Datei auch mehrmals in der Bibliothek steht.

Version 2.0

In Version 2.0 besteht der Bibliothekskopf aus ASCII-Zeichen, bei den Versionen 1.0B und 1.0C aus Binärzeichen, die Sie nicht lesen können.

Bibliotheken der Version 2.0 und der Versionen 1.0B/1.0C sind daher untereinander nicht kompatibel. Die Bibliotheken müssen von dem ar-Kommando der Version erzeugt worden sein, mit der Sie arbeiten. Sie können z.B. keine Bibliothek der Version 1.0C benutzen, wenn Sie mit Version 2.0 arbeiten.

Bibliotheken müssen ein Inhaltsverzeichnis enthalten, wenn sie mit Objektdateien zusammengebunden werden sollen (von cc oder ld). Im Inhaltsverzeichnis stehen nur die Namen der Bibliotheksdateien, die Objektmodule sind. Am besten behandeln Sie jede Bibliothek, die Sie mit ar bearbeiten, sofort danach mit ranlib. Ein "makefile" (siehe Kommando "make", Betriebssystem SINIX, Buch 1) kann diese Aufgabe für Sie übernehmen.

Beispiele

1. Eine Bibliothek erstellen:

```
$ ar vq archiv.a atoi.o itoa.o
ar: creating archiv.a
q - atoi.o
q - itoa.o
$
```

   Bibliotheksnamen enden üblicherweise mit ".a". Falls *archiv.a* noch nicht existiert, wird es erstellt und ar kopiert die Dateien *atoi.o* und *itoa.o* nach *archiv.a.* Existiert *archiv.a* bereits, werden die beiden Dateien am Ende der Bibliothek angefügt. ar prüft nicht, ob die Dateien schon in der Bibliothek stehen. Die Angabe von v bewirkt, daß erläutert wird, was jeweils geschieht.

2. Eine neue Datei an einer angegebenen Stelle in eine Bibliothek einfügen:

```
$ ar rb atoi.c archiv.a atof.c
$
```

   ar kopiert *atof.o* vor *atoi.o* ins Archiv.

3. Das Inhaltsverzeichnis der Bibliothek ausgeben lassen:

```
$
rw-rw-r-- 27/1      38 Jul 17 11:49 1985 atof.o
rw-rw-r-- 27/1      38 Jul 17 11:49 1985 atoi.o
rw-rw-r-- 27/1      38 Jul 17 11:49 1985 itoa.o
$
```

4. Eine Datei aus der Bibliothek kopieren:

```
$
$
```

Die Bibliotheks-Datei *atoi.o* wird in das aktuelle Dateiverzeichnis kopiert. Der Name der Kopie ist auch *atoi.o*. Soll die Kopie den Namen *copyatoi.o* erhalten, sieht der ar-Befehl folgendermaßen aus:

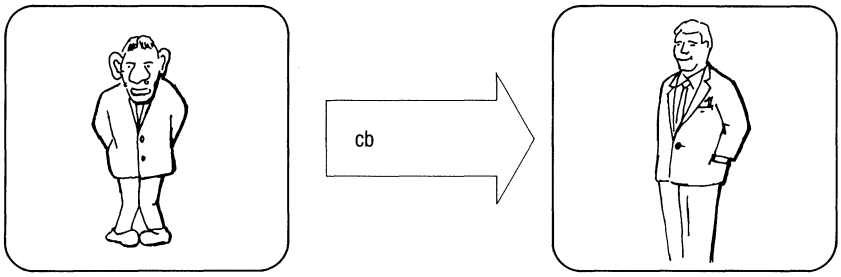
```
$
$
```

Dateien

/tmp/v* Zwischendateien

> > > > ld, lorder, ranlib

C-Programme formatieren - C program beautifier



cb formatiert ein C-Quell-Programm.
Das C-Programm sollte danach "schöner" und leichter lesbar sein.

Die Eingabe erfolgt über die Standard-Eingabe. cb stellt das Ergebnis auf der Standard-Ausgabe zur Verfügung. cb verändert nicht den Text, nur die äußere Form.

cb

Hinweis

- Das Ergebnis entspricht manchmal nicht den Erwartungen.
- Durch Einrückungen können überlange Zeilen entstehen. Sie bekommen z.B. Schwierigkeiten, wenn Sie eine Datei, die zu lange Zeilen hat, mit ced bearbeiten wollen.

Version 2.0

Die Beschreibung von cb ist für alle Versionen (1.0B, 1.0C, 2.0) gültig.

Beispiel

```
$
$
```

Ein C-Programm wird aus der Datei *prog.c* "verschönert" in die Datei *progschoen.c* geschrieben.

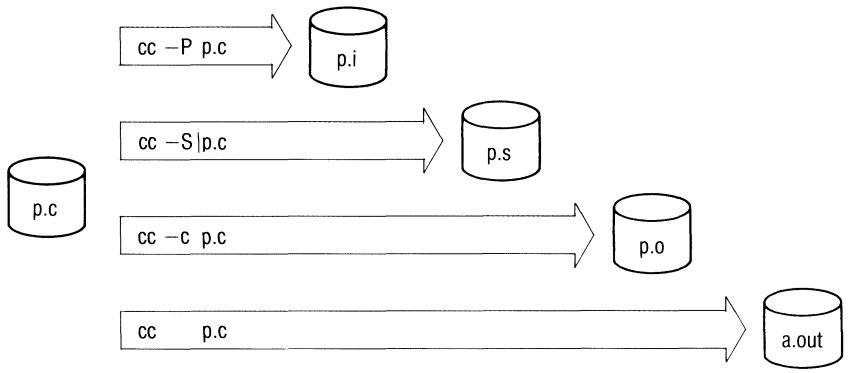
Inhalt von *prog.c*:

```
#define ENDE 20
main()
{int i;for(i=0;i<ENDE;i++){printf
("Das Quadrat von %d lautet %d\n",i,i*i);}}
```

Inhalt von *progschoen.c*:

```
#define ENDE 20
main()
{
        int i;
        for(i=0;i<ENDE;i++){
                printf
                    ("Das Quadrat von %d lautet %d\n",i,i*i);
        }
}
```

Steuerprogramm zum Übersetzen von C-Programmen - C compiler



cc ist ein Steuerprogramm zum Übersetzen von C-Programmen.
cc kann C-Quell-Programme und Objektmodule bearbeiten.

Wenn keine Fehler beim Übersetzen oder Binden aufgetreten sind, steht in der Datei a.out ein ablauffähiges Programm (siehe auch Schalter -o).

cc[␣schalter...]␣datei...

schalter

-c cc führt nur die ersten vier Schritte (siehe unten) aus und erstellt Objektmodule. Die .o-Datei wird auch dann nicht gelöscht, wenn Sie nur ein einziges C-Programm an cc übergeben (siehe 4.).
Z.B steht ein Objektmodul, der aus der Datei *hans.c* erzeugt wird, in der .o-Datei *hans.o*.

-p cc erzeugt einen Code, der mitzählt, wie oft jede Funktion aufgerufen wird. Außerdem wird beim Binden die Standard-Startfunktion ersetzt durch eine Funktion, die beim Starten automatisch die Funktion monitor aufruft.

Wenn ein Programm normal beendet wird, steht die Zählinformation in der Datei mon.out (siehe monitor).

Mit dem prof-Kommando können Sie mon.out auswerten und eine Tabelle der CPU-Zeiten erstellen (siehe prof). prof kann keine Auswertung vornehmen, wenn Sie gleichzeitig den Schalter -s (siehe ld) angeben.

-O Im dritten Schritt optimiert cc den Code, der im zweiten Schritt erzeugt wird. Der Schalter -O hilft Rechenzeit und Speicherplatz sparen. Sie sollten den Schalter nur bei einem Programm einschalten, das fehlerfrei läuft. Die Übersetzungszeit erhöht sich nämlich, wenn Sie diesen Schalter angeben.

-S Assembler-Quell-Programme, die cc im zweiten Schritt erzeugt, werden in entsprechende Dateien geschrieben, die mit ".s" enden. cc führt keine weiteren Schritte aus.

Z.B. steht der Assembler-Quell-Code, der aus einem C-Quell-Programm in *datei.c* erzeugt wird, in der Datei *datei.s*.

-P .c-Dateien werden dem Makro-Präprozessor übergeben, d.h. cc führt nur den ersten Schritt aus. Das Ergebnis wird in die entsprechenden .i-Dateien geschrieben. .i-Dateien enthalten keine Zeilen, die mit "#" beginnen.

Mit diesem Schalter können Sie überprüfen, ob die Präprozessoranweisungen im C-Quell-Programm auch das machen, was Sie möchten.

-E cc behandelt .c-Dateien ähnlich wie beim Schalter -P. Das Ergebnis wird allerdings auf der Standard-Ausgabe ausgegeben. Präprozessoranweisungen werden nicht vollständig entfernt; das Ergebnis kann noch Zeilen enthalten, die mit "#" beginnen.

-o␣ausgabe

cc schreibt ausführbaren Programmcode in die Datei *ausgabe* statt nach a.out. a.out bleibt unverändert bzw. wird nicht erstellt.

Bricht cc bereits nach vier Schritten ab, so wird der erzeugte Objektmodul in die Datei *ausgabe* geschrieben, statt in eine .o-Datei.

-**D**name[=def]

name wird definiert, wie bei einer #define-Anweisung in einem C-Programm, und vom Präprozessor genauso behandelt. Im C-Quell-Programm könnte statt diesem Schalter stehen:

```
#define name def
```

Fehlt die Definition *def*, so wird *name* als 1 definiert.

-**U**name Definitionen, bei denen *name* initialisiert wurde, werden entfernt. Im C-Quell-Programm könnte stattdessen stehen

```
#undef name
```

-**I**dir *dir* ist ein Dateiverzeichnis.

cc sucht nach Include-Dateien, deren Name nicht mit "/" beginnt und deren Name von "<" und ">" eingeschlossen wird,

- zuerst im Dateiverzeichnis der Dateien, die Sie als *datei* angeben,
- dann in den Dateiverzeichnissen *dir* (der Schalter -I kann mehrmals angegeben werden),
- dann im Dateiverzeichnis /usr/include.

-Xdir *dir* ist ein Dateiverzeichnis.

cc sucht nach Include-Dateien, deren Name nicht mit "/" beginnt und deren Name von "<" und ">" eingeschlossen wird, wie beim Schalter -I,

- zuerst im Dateiverzeichnis der Dateien, die Sie als *datei* angeben,
- dann in den Dateiverzeichnissen *dir* (der Schalter -X kann mehrmals angegeben werden),

aber nicht

- im Dateiverzeichnis /usr/include.

schalter des ld-Befehls

Siehe ld.

Die meisten Schalter für den ld-Befehl können Sie beim Aufruf von cc angeben. cc übergibt beim fünften Schritt diese Schalter an ld. Die Schalter -X und -P werden nicht an ld übergeben.

datei *datei* ist ein Dateiname. Dateinamen sollten die Endung haben, die cc erwartet. cc erkennt aus dem Dateinamen, welche Programme eine Datei enthält und führt die jeweils noch erforderlichen Übersetzungsschritte aus.

Die Dateien können sein:

- C-Quell-Programme (Endung ".c"),
- Assembler-Quell-Programme (Endung ".s"),
- Bibliotheken mit C-kompatiblen Funktionen, die bei früheren cc-Aufrufen erzeugt wurden (siehe auch ld, Schalter -l) (Endung ".a"),
- Objektmodule, die bei früheren cc- oder ld-Aufrufen erzeugt wurden (Endung ".o").

Um ein ausführbares Programm zu erstellen, werden die Dateien in der angegebenen Reihenfolge bearbeitet und gebunden.

Wie arbeitet cc?

cc bearbeitet ein C-Quell-Programm in fünf Schritten. Das Ergebnis eines Schritts wird dabei jeweils dem nächsten Schritt zur Verfügung gestellt:

1. *Präprozessor-Anweisungen durchführen*

   Der Präprozessor cpp behandelt Makros und fügt Include-Dateien ein. Es findet nur eine textuelle Ersetzung statt. Sämtliche Zeilen eines C-Quell-Programms, die mit "#" beginnen, werden entfernt und es werden die Aktionen ausgeführt, die nach "#" angegeben sind.

   Standard-Include-Dateien stehen normalerweise im Dateiverzeichnis /usr/include. Ihre Namen enden nach Konvention mit ".h", z.B. stdio.h. Im C-Programm stehen die Namen zwischen den Klammern "<" und ">", z.B.

   ```
   #include <stdio.h>
   ```

2. *Übersetzen (in Assembler-Quell-Code)*

   Der C-Quellcode wird in Assembler-Quellcode übersetzt. Siehe auch Schalter -S. Der Assembler-Quellcode eines C-Programms *hans.c* steht in *hans.s.* Wenn cc weitere Schritte ausführt, wird die .s-Datei wieder gelöscht.

3. *Optimieren*

   Um Rechenzeit und Speicherplatz zu sparen, kann der Assembler-Quellcode optimiert werden.
   Siehe Schalter -O.

4. *Assemblieren*

   cc ruft den Assembler as auf. Assemblierte Programme, sog. Objektmodule, werden in Dateien abgelegt, die mit ".o" enden. Z.B. werden C-Programme, die in den Dateien *jiri.c* und *guy.c* stehen, assembliert in die Dateien *jiri.o* und *guy.o* geschrieben.

   cc löscht die .o-Datei, wenn nur ein einziges C-Programm fehlerfrei übersetzt und sofort gebunden wird.

5. *Binden*

   cc ruft das Kommando ld auf. Schalter für ld können Sie beim cc-Aufruf angeben. ld schreibt sein Ergebnis in die Datei a.out (siehe auch Schalter -o).

Wenn cc diese fünf Schritte fehlerfrei durchlaufen hat (und nicht der Schalter -o eingeschaltet ist), enthält die Datei a.out ausführbaren Programmcode.

Hinweis

- Wird an cc ein Schalter übergeben, der cc veranlaßt nach weniger als fünf Schritten abzubrechen, so ignoriert cc alle Schalter, die sich auf Schritte nach dem Abbruch beziehen.

  ```
  $ cc -o covalam -S covalam.c
  ```

  Der Schalter -o wird ignoriert. Der erzeugte Assembler-Quellcode steht in der .s-Datei *covalam.s*.

- Wenn cc auf dem Prozessor 8086/80186 und mit dem System UNIX/SINIX läuft, dann setzt cc die Schalter -DI8086 und -Dunix. Sie können z.B. abfragen, auf welchem Prozessor ein Programm übersetzt wird.

- Beim Übersetzen werden die ersten 7 Zeichen von Variablen- und Funktionsnamen unterschieden. Z.B. bezeichnet "anzeiger1" dieselbe Variable wie "anzeiger2".

Fehlerdiagnose

Beim Auftreten eines Fehlers werden ausgegeben:

- der Name der Datei, in der der Fehler auftritt,
- die Nummer der Zeile, in der der Fehler bemerkt wird,
- eine kurze Fehlerbeschreibung.

Nach der ersten Fehlermeldung sind die Angaben zu Folgefehlern oft nicht korrekt. Sie sollten deshalb zuerst die ersichtlichen Fehler beseitigen und das Programm noch einmal übersetzen.

Fehlerdiagnosen, die cc ausgibt, können auch vom Assembler oder Binder stammen.

Version 2.0

- Beim Übersetzen werden die ersten 15 Zeichen in Variablen- und Funktionsnamen unterschieden.
- Die Bibliothek /lib/libffp.a ist in der Bibliothek /lib/libc.a enthalten. cc bindet im 5. Schritt automatisch /lib/libc.a dazu. Der Aufruf in Beispiel 2 kann daher lauten:

  ```
  $ cc prog.c -lm
  ```

- Schalter, die in Version 2.0 zusätzlich vorhanden sind:

  -C Der Präprozessor entfernt nicht die Kommentare im C-Quellcode.

  -V cc und die Programme, die cc aufruft (z.B. c1, c3, ld), kommentieren, was sie gerade ausführen.

  -**W**[n] Der Schalter legt den Umfang der Warnungen fest. Wenn *n* fehlt, gilt standardmäßig : $n = 1$.

  n kann folgende Werte annehmen:

0	cc unterdrückt alle Warnungen.
1	cc gibt nur wenige Warnungen aus.
2	cc gibt alle Warnungen aus.

  Programme, die cc aufruft, geben auch weiterhin Warnungen aus.

- Version 2.0 kennt keinen Optimierer. Der Schalter -O hat deshalb keine Wirkung. Datei /lib/c2 (der Optimierer) fehlt bei Version 2.0.
- Der Parser und der Code Generator sind aufgeteilt und stehen in Datei /lib/c1 und Datei /lib/c3.
- In Version 2.0 hat der Schalter -X eine andere Bedeutung als in Version 1.0B/1.0C.

  -X cc führt dasselbe wie bei Schalter -S aus. Zusätzlich zum Assembler-Quell-Code enthält die .s-Datei den C-Quellcode als Kommentar.

- In Version 2.0 dürfen Sie folgende Schalter des ld-Befehls beim cc-Aufruf angeben:

  -G, -i, -n, -o, -s und -x.

Beispiele

1. Ein C-Programm übersetzen und binden:

```
$
$
```

cc übersetzt das C-Programm in *ladakh.c* und erstellt eine ausführbare Datei *ladakh*.

Sie können das C-Programm aufrufen mit:

```
$
```

2. Mathematische Funktionen in C-Programmen verwenden:

Das folgende C-Programm verwendet die mathematische Funktion power. In der Datei /usr/include/math.h werden die mathematischen Funktionen als extern deklariert. Mit "#include <math.h>" wird diese Datei in das C-Programm eingebettet.

Der Objektcode für die so deklarierten Funktionen muß anschließend an das übersetzte Programm angebunden werden.

In der Bibliothek /lib/libm.a steht der Code für die mathematischen Funktionen. In /lib/libffp.a stehen die Funktionen, die für Gleitkomma-Arithmetik nötig sind. Der Binder ld erlaubt für die Bibliotheken Abkürzungen, hier -lm bzw. -lffp (siehe auch ld Schalter -l).

Es ergibt sich folgender Aufruf für die Übersetzung:

```
$
$
```

Programm in Datei *prog.c*:

```
#include <math.h>        /* Deklaration der externen math. Funktionen */
#define ENDE 20
#define QUADRAT 2.0

main()
{
    double i;
    for(i=0;i<ENDE;i++){
        printf("Das Quadrat von %.0f lautet %.0f\n",i,pow(i,QUADRAT));
    }
}
```

Dateien

datei.c	Eingabedatei, die ein C-Programm (C-Programme) enthält
datei.o	Objektmodul
a.out	enthält ausführbaren Programm-Code nach Ablauf des cc-Kommandos
/tmp/t[123]*	Zwischendateien
/lib/cpp	Präprozessor
/lib/c0	Übersetzer für cc Lauf 1 (Scanner)
/lib/c1	Übersetzer für cc Lauf 2 (Parser, Code Generator)
/lib/c2	Optimierer
/lib/crt0.o	Laufzeit-Start-Funktion
/lib/mcrt0.o	Laufzeit-Start-Funktion mit monitor-Aufruf
/lib/libc.a	Standard-Bibliothek
/usr/include	Standard-Dateiverzeichnis für Include-Dateien.

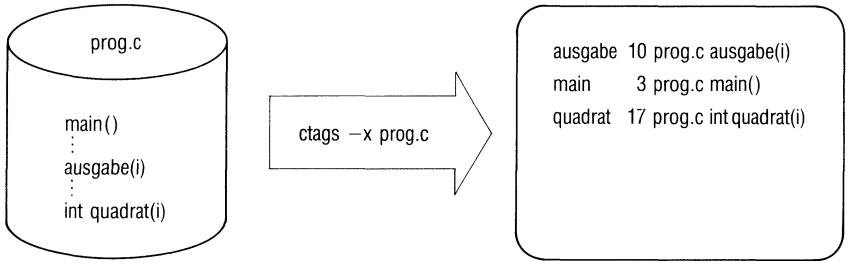
Siehe auch

B.W.Kernighan and D.M.Ritchie,
Programmieren in C,
Hanser Verlag München, 1984
deutsche Ausgabe von Prof.Dr.A.T.Schreiner, Dr.E.Janich

D.M.Ritchie,
C Reference Manual

> > > > adb, ld, monitor, prof

Tags-Datei erstellen - create a tags file



ctags hilft Ihnen, Funktionsdefinitionen in C-Quell-Programmen zu finden.

ctags erstellt für die angegebenen Dateien, in denen C-Quellcode stehen sollte, eine tags-Datei. Der Name dieser tags-Datei lautet "tags".

Eine tags-Datei gibt an, wo bestimmte Objekte (hier: Funktionen) in den entsprechenden Dateien stehen. Eine Zeile einer tags-Datei enthält folgende Informationen:

- den Funktionsnamen,
- den Namen der Datei, in der die Funktion definiert wird,
- ein Textmuster, das Sie zum Suchen der Funktionsdefinition mit einem Editor, wie z.B. ced, verwenden können (ced -t ...).

ctags[␣-schalter...]␣datei...

schalter

a — ctags hängt die Ausgabe an das Ende der Datei tags an. Die Datei tags wird also, falls sie schon existiert, nicht überschrieben.

u — ctags bringt die Informationen über die angegebenen Dateien auf den neuesten Stand. D.h. in der tags-Datei werden die alten Informationen gelöscht und die neuen in die Datei tags eingefügt.

Vorsicht!

ctags arbeitet ziemlich langsam, wenn dieser Schalter eingeschaltet ist. Meistens geht es schneller, die tags-Datei noch einmal neu zu erstellen.

w — Warnungen von ctags werden unterdrückt.

x — ctags schreibt nicht in die tags-Datei. Stattdessen gibt ctags auf der Standard-Ausgabe eine Liste mit Informationen über die angegebenen Dateien aus.

Eine Listenzeile enthält folgende Angaben:

- den Funktionsnamen,
- die Zeilennummer, in der die Funktion definiert wird,
- den Dateinamen der Datei, in der die Funktion definiert wird,
- den Text der Definitionszeile.

ctags unterläßt die Sonderbehandlung des Funktionsnamens main (siehe unten).

Wenn Funktionsnamen mehrmals vorkommen (z.B. main), gibt ctags eine Warnung aus.

datei — In der Datei *datei* sollten C-Quellprogramme stehen und *datei* sollte ein Name sein, der mit ".c" oder mit ".i" endet. ctags durchsucht die Datei nach Definitionen von C-Funktionen und C-Makros.

Sonderbehandlung des Funktionsnamens main

ctags verwendet statt main den Dateinamen der Datei, in der main definiert ist. Diesem Dateinamen wird der Buchstabe "M" vorangestellt. ctags entfernt ein eventuell vorhandenes ".c" oder ".i" am Namensende. Pfadnamenkomponenten, die vor dem Dateinamen stehen, werden ebenfalls entfernt.

Z.B. wird ein Programm main in der Datei */usr/jannis/help/programm.c* in der tags-Datei mit *Mprogramm* bezeichnet.

Diese Sonderbehandlung von main erweist sich als praktisch, wenn mit ctags eine tags-Datei für Dateiverzeichnisse erstellt wird, die mehr als ein Hauptprogramm main enthalten.

Wenn der Schalter x eingeschaltet ist, unterläßt ctags die Sonderbehandlung von main.

Hinweis

Falls Sie Dateien angeben, die keine C-Quellprogramme enthalten, kann der Text, den ctags in die tags-Datei schreibt, unverständlich sein.

Version 2.0

Die Beschreibung von ctags ist für alle Versionen (1.0B, 1.0C, 2.0) gültig.

Beispiel

In der Datei *programm.c* steht folgendes C-Programm:

```
#define ENDE 20

main()
{
        int i;
        for(i=0;i<ENDE;i++)
                ausgabe(i);
}

                /* Ausgabe von i und i*i */
ausgabe(i)
int i;
{
        printf("Das Quadrat von %d lautet %d\n",i,quadrat(i));
}

                /* Berechnung von i*i */
int quadrat(i)
int i;
{
        return(i*i);
}

$
```

ctags gibt auf der Standard-Ausgabe aus:

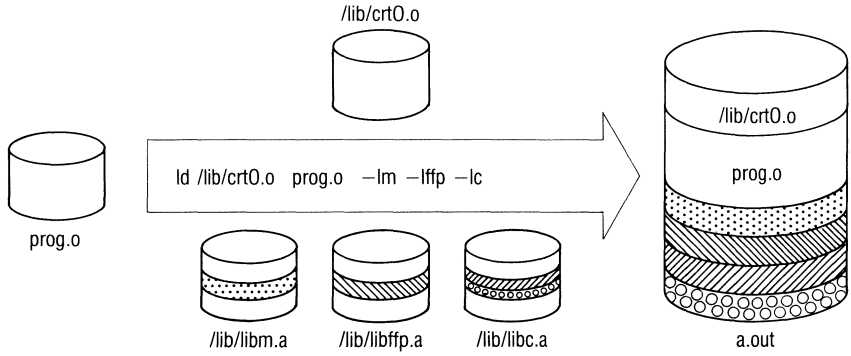
```
ausgabe      10 programm.c       ausgabe(i)
main          3 programm.c       main()
quadrat      17 programm.c       int quadrat(i)
$
```

Dateien

ctags	ist die tags-Datei, die ctags erstellt.

> > > > ced

Binder aufrufen - loader



ld

- bindet mehrere Objektdateien zu einer (Ausgabe-)Objektdatei zusammen,
- löst externe Referenzen (Variablen und Funktionen) auf, d.h.
- sucht in Bibliotheken nach Objektmodulen, in denen Funktionen oder Variablen definiert werden, deren Definition noch nicht erfolgte und
- bindet die Objektmodule mit den passenden Definitionen an die Ausgabedatei.

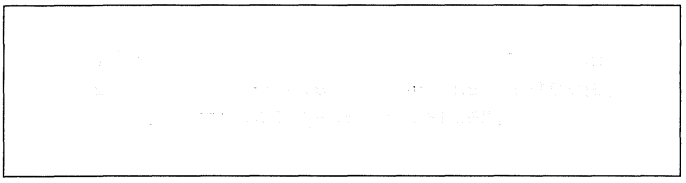
Im einfachsten Fall bindet ld mehrere Objektdateien zu einer Objektdatei zusammen.

Diese Objektdatei kann

- ausgeführt werden, wenn beim Binden keine Fehler auftreten,
- als Eingabe für einen weiteren ld-Aufruf dienen. In diesem Fall muß der Schalter -r gesetzt sein. ld kann mehrere Objektdateien, die bereits zu einer zusammengebunden sind, schneller weiterverarbeiten, als wenn Sie die Dateien einzeln an ld übergeben.

ld schreibt die Ausgabe in die Datei a.out, wenn nicht der Schalter -o eingeschaltet ist.

cc ruft im 5.Schritt das ld-Kommando auf.



ld[␣schalter...]␣datei...[␣-lxyz...]

schalter — Außer dem Schalter -l sollten die Schalter, wie üblich, vor den Dateinamen stehen.

-s — ld entfernt die Symboltabelle und die Relokationsbits von der Ausgabe. So wird Externspeicherplatz (nicht Kernspeicherplatz) gespart, aber die Fehlersuche wird schwieriger. Sie können dann z.B. bei adb-Kommandos keine symbolischen Adressen verwenden.

ld mit dem Schalter -s hat dieselbe Wirkung wie strip.

-u␣arg Das Argument *arg* wird als undefiniertes Symbol in die Symboltabelle eingetragen. ld versucht den Modul mit der Definition des Symbols *arg* zu finden und dazuzubinden. Dieser Schalter erlaubt es, Module aus Bibliotheken zu binden, die sonst nicht dazugebunden werden. *arg* kann z.B. ein Funktionsname sein.

-lxyz Der Schalter ist eine Abkürzung für die Bibliothek

/lib/lib*xyz*.a, wobei *xyz* eine Zeichenreihe ist. Wenn diese Bibliothek nicht existiert, versucht ld die Bibliothek /usr/lib/lib*xyz*.a zu finden. ld sucht nur dann nach einer Bibliothek, wenn ihr Name angegeben ist.
Sie können mehrere Bibliotheken (in der richtigen Reihenfolge!!!) angeben.

ld durchsucht eine genannte Bibliothek, um externe Referenzen aufzulösen. ld sucht z.B. nach der Definition einer Funktion, die bereits aufgerufen, aber noch nicht definiert wurde. Wenn die Definition in der Bibliothek steht, bindet ld den Objektmodul mit der Definition an die Ausgabedatei.

Je nachdem, welche Funktionen Sie in den angegebenen Objektdateien verwenden, müssen Sie die entsprechenden Bibliotheken beim ld-Aufruf (bzw. beim cc-Aufruf) angeben.

Beispiel

```
-lm             ld durchsucht /lib/libm.a
-lffp           ld durchsucht /lib/libffp.a
```

-x ld nimmt keine lokalen Symbole in die Symboltabelle auf. Die Ausgabedatei wird kleiner, die Fehlersuche wird schwieriger. Wenn ld von cc aufgerufen wird, dann ist dieser Schalter standardmäßig eingeschaltet.

-X Die lokalen Symbole bleiben in der Symboltabelle stehen. Nur lokale Symbole, die mit "L" beginnen, also intern generierte Marken, werden entfernt.

-r ld erzeugt Relokationsbits in der Ausgabedatei, so daß sie als Eingabe für weitere ld-Aufrufe verwendet werden kann. Mögliche Referenzen löst ld auf, common-Symbole bleiben undefiniert und undefinierte Symbole ergeben keine Fehlermeldung.

ld kann bei einem späteren ld-Aufruf Dateien, die mit -r zu einer Datei zusammengebunden sind, schneller weiterverarbeiten.

Wird dieser Schalter an den cc-Befehl übergeben, so wird die vom Assembler erstellte .o-Datei nicht gelöscht.

-d ld erzwingt die Definition von common-Symbolen sogar dann, wenn der Schalter -r eingeschaltet ist.

-n Wenn die Ausgabedatei (z.B. a.out) ausgeführt wird, sollen das (Programm)-Textsegment und das Datensegment getrennt werden und in verschiedenen Adreßräumen liegen.

Das Textsegment (auch "Codesegment" oder "Befehlsteil" genannt) ist dann schreibgeschützt und kann z.B. nicht als Folge eines Programmierfehlers überschrieben werden. Wenn die Ausgabedatei von mehreren Benutzern gleichzeitig ausgeführt wird, teilen sich diese den Textbereich und das Textsegment ist nur einmal im Hauptspeicher vorhanden.

-i Dieser Schalter bewirkt genau dasselbe wie der Schalter -n.

-**o**␣ausgabe

ld soll die Ausgabe in die Datei *ausgabe* schreiben, und nicht nach a.out.

-**e**␣arg Die Adresse des Symbols *arg* soll als Startadresse für das ausführbare Programm genommen werden.

Standard (keine Angabe):
Die Adresse 0 ist die Startadresse.

-D␣z	Das Datensegment soll die Größe *z* haben. *z* muß eine Dezimalzahl sein. Dieser Schalter kann nicht über den cc-Befehl an den Binder übergeben werden, denn cc hat selbst einen Schalter -D, der eine andere Bedeutung hat.
datei	ld erwartet als *datei* den Namen eines Objektmoduls, der bei einem früheren cc-Aufruf oder bei einem früheren ld-Aufruf erstellt wurde.

Wie arbeitet ld?

- ld bindet die Dateien in der angegebenen Reihenfolge.
- Zu den genannten Objektmodulen bindet ld noch die Objektmodule aus Bibliotheken, die externe Referenzen auflösen. ld sucht in einer Bibliothek nach Objektmodulen genau ab dem Punkt, an dem die Bibliothek in der Argumentenliste des ld-Aufrufs steht.
- ld zieht die Dateien und Bibliotheken, die schon abgearbeitet sind, nicht mehr zur Auflösung der externen Referenzen heran.
  ld bindet nur die Module aus einer Bibliothek, die externe (noch nicht aufgelöste) Referenzen definieren.
- Allgemein gilt:

  Wenn Funktion *a* Funktion *b* aufruft und der Objektmodul mit der Definition von *a* *a.o*, der mit der Definition von *b* *b.o* heißt, dann muß

  - entweder *a.o* vor *b.o* in derselben Bibliothek stehen
  - oder die Bibliothek ein Inhaltverzeichnis haben (siehe ranlib)
  - oder die Bibliothek mit *a.o* muß vor der Bibliothek mit *b.o* in der Argumentenliste des ld-Aufrufs angegeben werden.

  Die Reihenfolge, in der Sie die Objektmodule, die Bibliotheken und die Funktionen angeben und definieren, ist also sehr wichtig.
- Die Reihenfolge der Funktionen innerhalb einer Bibliothek ist nur dann beliebig, wenn Sie die Bibliothek mit ranlib vorbehandeln, d.h. wenn die Bibliothek ein Inhaltsverzeichnis hat. Das Inhaltsverzeichnis heißt "_ _.SYMDEF". Es ist der erste Eintrag in der Bibliothek.
  Dieses Inhaltsverzeichnis wird so lange durchsucht, bis alle noch nicht aufgelösten Referenzen soweit wie möglich abgearbeitet sind.

Hinweis

- Die Symbole "_etext", "_edata" und "_end" ("etext", "edata" und "end" in C) sind reserviert und beziehen sich auf die erste Stelle nach dem Programmtext, nach den initialisierten Daten bzw. nach allen Daten. ld meldet einen Fehler, wenn Sie diese Symbole in einem Programm definieren.
- Je nachdem welche Eigenschaften die Objektmodule haben, die Sie binden wollen, müssen Sie entspechende Bibliotheken und Objektmodule beim ld-Aufruf zusätzlich angeben. Die wichtigsten Objektmodule und Bibliotheken (Schalter -l) lauten:

Welche Eigenschaften haben die Objektmodule, die Sie binden wollen?	Welchen Modul bzw. welchen Schalter -l müssen Sie beim ld-Aufruf zusätzlich angeben?
Der Objektmodul wurde mit cc ohne den Schalter -p erzeugt. Um ein ablauffähiges Programm zu erstellen, müssen Sie die Laufzeit-Start-Funktion /lib/crt0.o dazubinden.	/lib/crt0.o
Der Objektmodul wurde mit cc -p erzeugt. Um ein ablauffähiges Programm zu erstellen, müssen Sie die Laufzeit-Start-Funktion /lib/mcrt0.o dazubinden.	/lib/mcrt0.o
Aufruf von mathematischen Funktionen aus der Bibliothek /lib/libm.a	-lm
- Verwendung von Gleitpunktzahlen, - Aufruf von Funktionen aus der Bibliothek /lib/libffp.a.	-lffp
Aufruf von Funktionen aus der Bibliothek /usr/lib/libtermcap.a	-ltermcap
- Aufruf von Funktionen aus der Bibliothek /lib/libc.a, - Operationen mit long Zahlen, - C-Hilfsfunktionen. Jeder C-Modul, aus dem ein ablauffähiges Programm erstellt werden soll, benötigt diese C-Hilfsfunktionen. Im Zweifelsfall sollten Sie den Schalter immer angeben.	-lc

- cc bindet im fünften Schritt automatisch die Laufzeit-Start-Funktionen /lib/crt0.o bzw. /lib/mcrt0.o und die Bibliothek /lib/libc.a dazu. Wenn Sie ld direkt aufrufen, müssen Sie diese Aufgabe selbst übernehmen.
- Die Reihenfolge, in der Sie die wichtigsten Bibliotheken und Funktionen angeben müssen, ist:

```
/lib/crt0.o /lib/mcrt0.o datei... -lm -lffp -ltermcap -lc
```

Version 2.0

- In Version 2.0 können Sie zusätzlich folgende Schalter angeben:

  -G Aus der Symboltabelle werden die nicht lokalen Symbole entfernt, die in anderen Modulen definiert sind.

  -m[datei]
  ld schreibt in die Datei *datei* eine Übersicht, die die absoluten Adressen von globalen Variablen und Funktionen nach dem Binden aufzeigt.

  Standard (keine Angabe):
  ld gibt die Übersicht auf der Standard-Ausgabe aus.

  -v ld kommentiert, was gerade ausführt wird. ld listet z.B. auf, welcher Modul aus einer Bibliothek geholt wird.

- In Version 2.0 werden das Text- und das Datensegment immer getrennt. Sie liegen in verschiedenen Adreßräumen. Die Schalter -i oder -n der Version 1.0B/1.0C sind automatisch immer eingeschaltet.
- Sie müssen alle Bibliotheken, die Sie beim ld-Aufruf angeben (Schalter -l), vorher mit ranlib behandeln. ld bearbeitet nur Bibliotheken mit Inhaltsverzeichnis "_ _.SYMDEF"
- Die Bibliothek /lib/libffp.a ist in der Bibliothek /lib/libc.a enthalten.
- Die Schalter -D, -d und -u existieren in Version 2.0 nicht.

Beispiele

1. Objektmodule für weiteren ld-Aufruf zusammenbinden:

```
$
$
```

ld bindet die Objektmodule *objekt1.o* und *objekt2.o* zu einem Objektmodul *objekt.o* zusammen.

2. Mathematische Bibliothek dazubinden:

In der Datei *prog.c* steht folgendes C-Programm:

```
#include <math.h>       /* Deklaration der externen math. Funktionen */
#include <stdio.h>

main()
{
        float  zahl;
        printf("Bitte Zahl eingeben: ");
        scanf("%f",&zahl);
        printf("LOG(%f)  = %f\n",zahl,log(zahl));
        printf("EXP(%f)  = %f\n",zahl,exp(zahl));
        printf("SQRT(%f) = %f\n",zahl,sqrt(zahl));
}
```

Bei Aufruf von:

```
$
$
```

schreibt der Assembler, der von cc im vierten Schritt aufgerufen wird, das assemblierte Programm in die Datei *prog.o*. Um ein ablauffähiges Programm zu bekommen, müssen noch Referenzen aufgelöst und Bibliotheken dazugebunden werden.

Der Binder muß so aufgerufen werden:

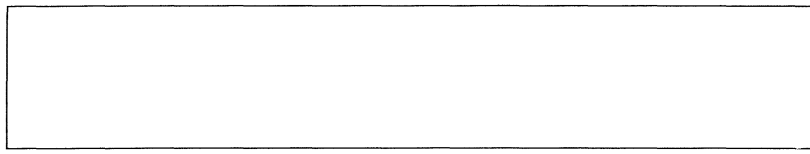
```
$
$
```

Die Laufzeit-Startfunktion /lib/crt0.o wird automatisch dazu gebunden, wenn der Übersetzer cc den ld-Aufruf übernimmt. Wird ld, wie in diesem Beispiel, vom Benutzer aufgerufen, muß diese Funktion vor dem eigenen Objektmodul gebunden werden. -lm und -lffp sorgen dafür, daß die mathematische Bibliothek und Gleitkomma-Arithmetik angebunden werden. -lc enthält C-Hilfsfunktionen (wird sonst auch von cc automatisch übergeben).

In a.out steht jetzt ein ablauffähiges Programm.

Der folgende ld-Aufruf ist **falsch** und führt zu einer Fehlermeldung:

```
$ ld -x /lib/crt0.o prog.o -lffp -lm -lc
Undefiniert:
_frexp
ltof
_end
$
```

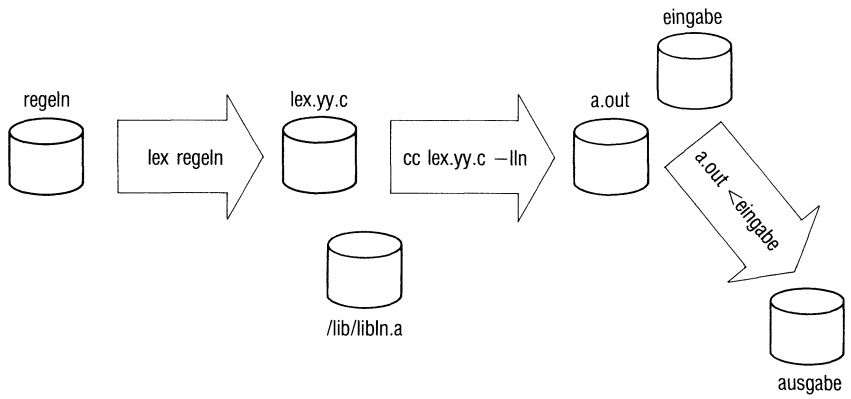


Dateien

/lib/lib*.a	Bibliotheken
/usr/lib/lib*.a	Bibliotheken, die der Benutzer selbst erstellen kann.
a.out	enthält die Ausgabe von ld.

> > > > ar, cc, ranlib, strip

Programme zur Textanalyse generieren - generator of lexical analysis programs



lex erstellt C-Programme, die einen Eingabetext analysieren und weiterverarbeiten. Die Analyse betrifft die einzelnen Zeichen, aus denen der Text besteht. Die Sprache des Eingabetextes muß eine Chomsky-3-Sprache (reguläre Sprache) sein.

Mit C-Programmen, die lex generiert hat, (**lex-Programme**) können Sie z.B.

- Eingabesprachen analysieren und prüfen, ob eine Eingabe syntaktisch korrekt ist (Plausabilitätsprüfung der Eingabe),
- Aktionen ausführen, wenn die Eingabe syntaktisch korrekt ist,
- (einfache) Textveränderungen vornehmen,
- eine Statistik erstellen über Zeichen oder Zeichenfolgen, die in einem Eingabetext vorkommen.

lex kann z.B. ein Programm generieren, das bei einem Übersetzer die lexikalische Analyse übernimmt. Ein solches Programm heißt **Scanner**. Der Scanner liest einen Eingabetext Zeichen für Zeichen ein und versucht, die Eingabe nach syntaktischen Einheiten vorzugruppieren.

Sie können insbesondere lex und yacc zusammen verwenden (siehe unten). Als Vorprozessor für ein Programm, das yacc generiert, unterteilt das lex-Programm einen Eingabetext in syntaktische Einheiten. Diese Einheiten heißen **Token** oder **Terminalsymbole**. Das yacc-Programm erkennt dann die Strukturen, zu denen die Token gehören, und veranlaßt entsprechende Aktionen.

Programme, die lex geschrieben hat, können also zusammen mit Programmen arbeiten, die von anderen Generatoren (wie z.B. yacc) oder vom Benutzer erstellt wurden.

lex[␣-schalter...][␣datei...]

schalter

t — lex schreibt das C-Programm auf die Standard-Ausgabe und nicht in die Datei lex.yy.c.

v — Zusätzlich zu dem Programm in lex.yy.c, erstellt lex eine Statistik (zum Programm), die auf der Standard-Ausgabe ausgegeben wird. lex erstellt die Statistik auch ohne Schalter v, wenn Sie im lex-Quell-Programm die Größe eines internen lex-Feldes neu definieren und nicht den Schalter n angeben (siehe unter "Definitionen in lex-Quell-Programmen").

n — Wenn Sie den Schalter v nicht angeben, können Sie den Schalter n setzen. Er bewirkt dasselbe, wie ein lex-Aufruf ohne Schalter v. lex erstellt keine Statistik.
Wenn Sie im lex-Quell-Programm die Größe eines internen lex-Feldes definieren, so müssen Sie den Schalter n angeben, um die Ausgabe einer Statistik zu unterdrücken (siehe unter "Definitionen in lex-Quell-Programmen").

Wenn Sie kein internes lex-Feld definieren, ist es z.B. egal, ob Sie lex aufrufen mit lex -fn oder mit lex -f.

f — lex erstellt ein Programm, das schneller übersetzt werden kann. Sie sollten diesen Schalter nur bei kleineren Programmen anwenden.

datei — *datei* ist der Name der Datei, in der das lex-Quell-Programm steht.

Standard (keine Angabe):
Standard-Eingabe.

Wie arbeitet lex?

lex erwartet als Eingabe ein lex-Quell-Programm, das einem bestimmten Format entsprechen muß (siehe unten). In dem lex-Quell-Programm geben Sie an:

- reguläre Ausdrücke, die Zeichenreihen (z.B. Token) beschreiben, nach denen der Scanner suchen soll;
- Aktionen, die in C geschrieben sind. Der Scanner soll eine bestimmte Aktion ausführen, wenn er eine Zeichenkette findet, die zu einem entsprechenden regulären Ausdruck paßt.

Ein regulärer Ausdruck zusammen mit den Aktionen, die bei einer passenden Zeichenreihe ausgeführt werden sollen, heißt **Regel**.

Aus den Informationen im lex-Quell-Programm erstellt lex ein C-Quell-Programm. lex schreibt es in die Datei lex.yy.c. In dem C-Programm wird u.a. eine Funktion yylex definiert, die das ausführt, was Sie in den Regeln angeben. Sie können Ein-/Ausgabefunktionen und ein Hauptprogramm, das yylex aufruft, entweder selbst definieren oder die Standardversionen der lex-Bibliothek /lib/libln.a verwenden.

Die lex-Bibliothek /lib/libln.a enthält folgende Objektmodule:

- allprint.o
- main.o
- reject.o
- yyless.o
- yywrap.o

Beim Übersetzen des Programms lex.yy.c müssen Sie entweder die lex-Bibliothek und/oder eigene Versionen der Funktionen oder des Hauptprogramms dazubinden. Der Aufruf zum Übersetzen lautet im ersten Fall:

```
$ cc lex.yy.c -lln
```

Wenn Sie eigene Versionen der lex-Bibliotheks-Funktionen bereitstellen und der Aufruf von yylex aus einem anderen Hauptprogramm erfolgt, müssen Sie die Bibliothek nicht dazubinden.

Wie arbeitet der Scanner?

Das übersetzte und gebundene Programm steht in a.out. Wenn Sie a.out ausführen, behandelt das Programm einen (auf der Standard-Eingabe bereitgestellten) Text, den **Eingabestrom**, folgendermaßen:

- es durchsucht den Eingabestrom nach Zeichenreihen, die zu den angegebenen regulären Ausdrücken passen und führt bei erfolgreicher Suche die entsprechenden Aktionen aus;
- es kopiert unverändert die Textteile des Eingabestroms, die nicht zu einem regulären Ausdruck passen, auf die Standard-Ausgabe.

Der Text, den a.out auf der Standard-Ausgabe erstellt, heißt **Ausgabestrom**.

Die Zeit, die ein lex-Programm zur Analyse eines Eingabestroms braucht, ist proportional zur Länge des Eingabestroms. Die Anzahl und Komplexität der Regeln beeinflußt nur dann die Zeit, wenn kontextabhängige Regeln vorkommen, die viel Zeit zum Vorauslesen und nochmaligen Analysieren von Textteilen benötigen.

Die Größe des lex-Programms wächst mit der Anzahl und Komplexität der Regeln.

In dem C-Programm, das lex erstellt, erzeugt lex aus den regulären Ausdrücken einen endlichen deterministischen Automaten. Der Automat befindet sich immer in genau einem Zustand. Der aktuelle Zustand und das Zeichen, das als nächstes eingelesen wird, bestimmen den Folgezustand. Wenn der Automat ein Token erkennt, befindet er sich in einem Endzustand. Ein Interpreter steuert den Ablauf, d.h. er legt fest wie der Automat von einem Zustand in einen anderen Zustand kommt. Die Aktionen werden wie die verschiedenen Fälle bei einer switch-Anweisung behandelt.

Beispiel für einen lex-Aufruf

Zusammenfassend sei hier noch einmal die Kommandofolge angegeben, mit der Sie lex aufrufen und einen Eingabestrom vom lex-Programm behandeln lassen. Das lex-Quell-Programm steht in der Datei *regeln*. Sie wollen die Funktionen der lex-Bibliothek benutzen. Der Eingabestrom steht in der Datei *eingabe*. Der Aufruf lautet:

```
$
$
$
```

Format eines lex-Quell-Programms

Das allgemeine Format eines lex-Quell-Programms sieht so aus:

```
[Definition...]
%%
Regel...
[%%]
[Benutzer-Funktion...]
```

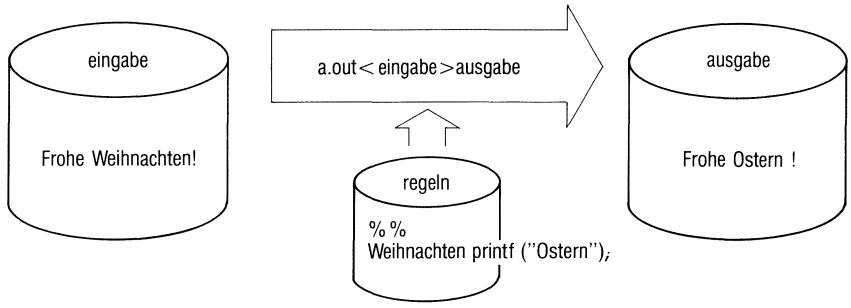
Definitionen und Regeln müssen im entsprechenden Format, Benutzer-Funktionen in C geschrieben sein.

Im Definitionsteil können Sie Definitionen für lex angeben. Außerdem können Sie im Definitionsteil oder im Regelteil vor der ersten Regel auch Text (Definitionen, Kommentare, etc.) an das C-Programm, das lex erstellt, übergeben.

Der Regelteil beschreibt die Zeichen oder Zeichenreihen (Token), die der Scanner im Eingabestrom erkennen soll.
Eine Regel besteht aus einem regulären Ausdruck und einer Aktion. Die Aktion ist ein C-Programmteil, der ausgeführt werden soll, wenn eine zum regulären Ausdruck passende Zeichenreihe gefunden wird.
Ein Tabulator- oder Leerzeichen kennzeichnet das Ende eines regulären Ausdrucks.
Besteht eine Aktion aus mehreren Anweisungen oder Deklarationen, so müssen Sie die Aktion in Klammern, "{" und "}", einschließen.

Benutzer-Funktionen sind vom Benutzer definierte C-Funktionen, die im lex-Programm aufgerufen werden können.

Beispiel



Das lex-Quell-Programm *regeln* besteht nur aus einer Regel. Statt "Frohe Weihnachten" im Eingabestrom wird im Ausgabestrom "Frohe Ostern" ausgegeben. Der restliche Eingabestrom wird unverändert als Ausgabestrom kopiert.

Welchen Text kopiert lex?

lex bearbeitet nur bestimmte Teile eines lex-Quell-Programms als eigentliche Definition für sich oder als Regel. Alle anderen Angaben im lex-Quell-Programm, kopiert lex nach lex.yy.c. Sie können auf diese Weise Kommentare ins lex-Programm (das Programm, das lex erstellt) einfügen oder Variablen für Aktionen definieren. Genauer gesagt kopiert lex folgenden Text ins Programm lex.yy.c:

- Jede Zeile im Definitionsteil oder im Regelteil vor der ersten Regel
  - die nicht Teil einer Regel oder Aktion ist und
  - die mit einem Tabulator- oder Leerzeichen beginnt,

  Sie können auf diese Weise z.B. Kommentare ins lex-Programm einfügen.

- Jeden Text im Definitionsteil oder im Regelteil vor der ersten Regel, der zwischen Zeilen steht, die nur

  ```
  %{
  und
  %}
  ```

  enthalten. Die Begrenzungszeilen werden nicht kopiert.

  Jede Zeile im Definitionsteil eines lex-Quell-Programms, die nicht mit Leer- oder Tabulatorzeichen beginnt, wird von lex als Definition (siehe unten) angesehen. Wenn Sie nun einen Text, der in der ersten Spalte beginnt, nach lex.yy.c kopieren wollen, müssen Sie den Text zwischen "%{" und "%}" schreiben.

  In der ersten Spalte müssen z.B. die Präprozessor-Anweisungen beginnen, die nach lex.yy.c kopiert werden sollen.

- Den gesamten Text, der nach dem zweiten "%%"-Zeichen steht ("Benutzer-Funktionen"). In diesem Abschnitt des lex-Quell-Programms kann der Benutzer u.a. Funktionen für die Aktionen oder für yylex definieren. Dabei ist es unwichtig, welches Format der Text hat.

Für den kopierten Text gilt

– lex-Quell-Code, der vor dem ersten "%%"-Zeichen steht, erscheint im lex-Programm als global zu allen Funktionen. Definitionen von globalen Variablen können so an das lex-Programm übergeben werden.

– lex-Quell-Code, der direkt nach dem ersten "%%"-Zeichen und vor der ersten Regel steht, wird als lokale Deklaration in den lex-Programm-Teil kopiert, der die Aktionen enthält (die Funktion yylex). Dieser lex-Quell-Code muß wie ein C-Programmteil ausschauen.

– Kommentare sollten den Regeln für Kommentare in C-Programmen entsprechen.

Definitionen in lex-Quell-Programmen

Im Definitionsteil eines lex-Quell-Programms können Sie

- Variablen und Funktionen für die Aktionen oder für Benutzer-Funktionen definieren oder deklarieren,
- Kommentare zum lex-Quell-Programm oder zum Code, den lex generiert, einfügen,
- allgemein: Text angeben, der ins lex-Programm kopiert werden soll,
- Definitionen für lex angeben, wie z.B.
  - Startzustände benennen,
  - Namen für reguläre Ausdrücke definieren,
  - Übersetzungstabellen bereitstellen,
  - Standard-Belegungen der Größe von lex-internen Tabellen ändern.

Definitionen für lex

lex betrachtet jede Zeile eines lex-Quell-Programms, die die folgenden Bedingungen erfüllt, als eigentliche Definition für sich, und nicht als Text, der kopiert werden soll:

- Die Zeile muß vor dem ersten "%%" Zeichen stehen.
- Die Zeile darf nicht zwischen Zeilen stehen, die nur "%{" bzw. "%}" enthalten.
- Der Text der Zeile muß in der ersten Spalte beginnen.

– Die Definitionen müssen in einem der folgenden Formate angegeben sein:

- name␣text

  name — *name* ist der Name der Definition. *name* muß mit einem Buchstaben beginnen.

  text — *text* ist ein beliebiger regulärer Ausdruck.

  Sie können die Zeichenreihe *text* in regulären Ausdrükken mit "{name}" ansprechen. Bei regulären Ausdrükken, die häufig in Regeln vorkommen, ist es sinnvoll, einen Namen für sie zu definieren.

  Als Trennzeichen zwischen *name* und *text* muß mindestens ein Leer- oder Tabulatorzeichen stehen.

- %**S**␣name1[[␣name2]␣name3] ... oder
  %**s**␣name1[[␣name2]␣name3] ... oder
  %**START**␣name1[[␣name2]␣name3] ...

  name1, name2, name3,...
  Sie definieren *name1*, *name2*, *name3* als Startzustände (siehe unter "Welche Möglichkeiten gibt es, Kontextabhängigkeit zu realisieren?").

- %T
  übersetzungstabelle
  %T

  übersetzungstabelle
  Normalerweise wird beim Ein- und Ausgabestrom die Ein- und Ausgabe mit den Funktionen input, output oder unput aus der lex-Bibliothek realisiert. Dabei wird ein Ein-/Ausgabe-Zeichen maschinenintern durch eine positive ganze Zahl repräsentiert.
  Wenn Sie z.B. eigene Ein-/Ausgabe-Funktionen bereitstellen und einzelne Zeichen durch andere Zahlen, als üblich, repräsentieren wollen, müssen Sie eine Übersetzungs-Tabelle angeben. Die Tabelle besteht aus Zeilen des Formats:

  zahl␣zeichenreihe
  Das Trennzeichen "␣" muß wieder aus mindestens einem Leeroder Tabulatorzeichen bestehen.

  zahl — *zahl* gibt die ganze Zahl an, durch die jedes Zeichen der Zeichenreihe *zeichenreihe* repräsentiert werden soll.
  zahl muß größer als 0 und kleiner als 128 sein.
  Sie müssen jedem Zeichen, das im Eingabestrom vorkommen kann, in der Übersetzungtabelle genau eine Zahl zuweisen.

  zeichenreihe
  zeichenreihe enthält alle die Zeichen, die durch den gleichen Zahlenwert *zahl* repräsentiert werden.

- %x␣zahl — Um größere lex-Quell-Programme zu bearbeiten, ist es oft nötig, die Standardgröße der Tabellen zu ändern, die lex intern benutzt.

  Wenn Sie im lex-Quell-Programm lex-Tabellen neu definieren, wird auf der Standard-Ausgabe eine Statistik ausgegeben (siehe Schalter v). Um diese Statistik zu unterdrücken, müssen Sie den Schalter n setzen.

  zahl — *zahl* ist eine positive ganze Zahl, die eine neue Größe für den Parameter festlegt, den *x* bezeichnet.

  x — *x* kann einer der folgenden Buchstaben sein:

Buchstabe	Parameter	Standardgröße
p	Positionen	2500
n	Zustände	500
e	Knoten von Bäumen	1000
a	Übergänge	2000
k	gepackte Zeichenklassen	1000
o	Ausgabefeldgröße	3000

Beispiel

Die Definition

```
%T
 1      Aa
 2      Bb
...     ...
26      Zz
27      \n
28       +
29       -
30       0
31       1
...     ...
39       9
%T
```

bildet Groß- und Kleinbuchstaben auf die Zahlen 1 bis 26, "Neue Zeile" auf die Zahl 27 usw. ab. Steht in einem lex-Quell-Programm diese Definition, dann dürfen nur die definierten Zeichen in einem Eingabestrom vorkommen und in den Regeln behandelt werden.

Regeln in lex-Quell-Programmen

Regeln müssen das folgende Format haben:

regulärerausdruck␣aktion

regulärerausdruck

regulärerausdruck ist ein regulärer Ausdruck (siehe unter "Reguläre Ausdrücke in lex-Quell-Programmen"), für den eine passende Zeichenfolge im Eingabestrom gefunden werden soll.

aktion — *aktion* ist der Teil eines C-Programms, der ausgeführt werden soll, wenn eine zum regulären Ausdruck passende Zeichenreihe gefunden wird. Besteht eine Aktion aus mehr als einer Anweisung oder Deklaration, so müssen Sie die Aktion in geschweifte Klammern, "{" und "}", einschließen (siehe unter "Aktionen in lex-Quell-Programmen").

␣ — Das Trennzeichen "␣" muß aus mindestens einem Trenn- oder Tabulatorzeichen bestehen.
lex betrachtet das erste Leer- oder Tabulatorzeichen, das in einer Regel auftritt, und nicht Teil eines regulären Ausdruck ist, als das Ende des regulären Ausdrucks.

Reguläre Ausdrücke in lex-Quell-Programmen

Ein regulärer Ausdruck r bezeichnet eine Menge von Zeichenfolgen, die passenden Zeichenfolgen (oder Interpretationen) von r.

Für die Suche nach einer Zeichenfolge, die zu einem regulären Ausdruck paßt, gelten folgende Regeln:

- Die Zeilen eines Textes werden von links nach rechts nach passenden Zeichenfolgen abgesucht.
- Es wird immer die längstmögliche passende Zeichenfolge genommen.

Ein regulärer Ausdruck besteht aus Textzeichen und Operatorzeichen. Ein Leerzeichen zeigt das Ende des regulären Ausdrucks an.

– Textzeichen sind die Zeichen, nach denen im Text gesucht werden soll.

– Operatorzeichen spezifizieren Wiederholungen, Auswahlmöglichkeiten, Fluchtsymbole usw.

Operatorzeichen sind:

" \ [ ] ^ - ? . * + | ( ) $ / { } % < >

Textzeichen sind:

– Alle Buchstaben und Ziffern.

– Alle Zeichen, die von Anführungsstrichen (") eingeschlossen sind.

- Jedes einzelne Zeichen, das sonst nicht unbedingt als Textzeichen interpretiert werden würde, wenn ihm ein Gegenschrägstrich ("\") vorausgeht, also z.B.

  \ Leerzeichen (ein Leerzeichen ohne " \" davor zeigt das Ende des regulären Ausdrucks an),
  \n Neue Zeile,
  \t Tabulatorzeichen,
  \b Rücktastenzeichen,
  \\ Gegenschrägstrich,
  \+ Pluszeichen.

- Zusammenfassend kann man sagen:

  Jedes Zeichen, außer dem Leerzeichen, dem Tabulatorzeichen, dem Zeichen für "Neue Zeile" und den Operatorzeichen ist ein Textzeichen.

Beispiel

Die regulären Ausdrücke

```
abc"++"
abc\+\+
"abc++"
```

bezeichnen dieselbe Textzeichenfolge, nämlich abc++. abc++ ist auch die einzige Zeichenfolge, die zu den regulären Ausdrücken paßt.

Reguläre Ausdrücke

Syntax	Interpretation	Beispiel	
Ein regulärer Ausdruck ist:	Der linksstehende reguläre Ausdruck bezeichnet	regulärer Ausdruck:	passende Zeichenfolge:
z z ein Text-zeichen	das entsprechende Text-zeichen	a	a
\z z ein Operator- oder Sonder-zeichen	das entsprechende Operator- oder Sonderzeichen	\{ \n	{ Neue Zeile
\a a ganze Zahl zwischen 0 und 177 (Programme mit regulären Ausdrücken, in denen \a vor-kommt, sind nicht immer portierbar)	das ASCII-Zeichen, das den Oktalwert a hat	\40	Leerzeichen
"s" s beliebige Zeichenfolge mit "\" als Fluchtsymbol	die Zeichenfolge s	"+␣ \""	+␣ "
[s] s beliebige Zeichenfolge, eventuell mit "\" als Flucht-symbol,"-" als Intervallsymbol (bei einem In-tervall a-b muß a<b sein nach der ASCII-Tabelle)	genau eines von den Zeichen, die in s vorkommen	[\t] ["ab] [+^] [-+0-9] [\[] [\40-\176]	Leer-oder Tabulator-zeichen a, b oder " + oder ^ Ziffern oder Plus- oder Minuszeichen [alle druck-baren ASCII-Zeichen

Syntax	Interpretation	Beispiel	
Ein regulärer Ausdruck ist:	Der linksstehende reguläre Ausdruck bezeichnet	regulärer Ausdruck:	passende Zeichenfolge:
[^s] s wie bei [s] ("^" muß direkt nach "[" stehen)	genau ein Zeichen, das nicht in s vorkommt	[^\\.&\"]	jedes Zeichen außer ", \ , . und &
.	ein beliebiges Zeichen aus dem Gesamtzeichenvorrat außer "Neue Zeile"	.	jedes Zeichen außer "Neue Zeile"
xy x, y reguläre Ausdrücke	eine passende Zeichenreihe für x, gefolgt von einer passenden Zeichenreihe für y	\+\+ abc	++ abc
r? r regulärer Ausdruck	eine oder keine passende Zeichenreihe für r (r optional)	ab?c	abc oder ac
r* r regulärer Ausdruck	eine Folge von 0,1,2,... passenden Zeichenreihen für r	m*	kein m oder m, mm, mmm,...usw.
r+ r regulärer Ausdruck	eine Folge von 1,2,3,... passenden Zeichenreihen für r	[a-z]+	alle Zeichenreihen aus Kleinbuchstaben
(r) r regulärer Ausdruck (zum Zusammenfassen von komplexen regulären Ausdrücken)	eine Zeichenreihe die zu r paßt	(abc) (ef)+ (ab)*[de]	abc ef, efef, efefef,...usw. d, e, abd, ababe,...usw.
x|y x, y reguläre Ausdrücke	entweder eine passende Zeichenreihe für x oder eine passende für y	(ab|cd+)?(ef)*	abefef, cdd efefef, cdef, ab,...usw.

Syntax	Interpretation	Beispiel	
Ein regulärer Ausdruck ist:	Der linksstehende reguläre Ausdruck bezeichnet	regulärer Ausdruck:	passende Zeichenfolge:
^r r regulärer Ausdruck ("^" muß vor dem gesamten regulären Ausdruck stehen, "^" innerhalb von "[" und "]" hat eine andere Bedeutung)	eine passende Zeichenreihe für r, die am Anfang einer Zeile vorkommt, also am Anfang des Eingabestroms oder nach dem Zeichen "Neue Zeile"	^abc	abc am Anfang einer Zeile
r$ r regulärer Ausdruck ("$" muß nach dem gesamten regulären Ausdruck stehen)	eine passende Zeichenreihe für r, die am Ende einer Zeile, also unmittelbar vor dem Zeichen "Neue Zeile", steht	Rand$	Rand am Ende einer Zeile
x/y x, y reguläre Ausdrücke	eine passende Zeichenreihe für x, wenn ihr unmittelbar ein passender Ausdruck für y folgt	Rand/\n	Rand am Ende einer Zeile
<c>r r regulärer Ausdruck, c ein Startzustand des Interpreters	eine passende Zeichenreihe für r, wenn der Interpreter des Automaten, der von lex generiert wird, im Startzustand c ist (siehe unter "Welche Möglichkeiten gibt es, Kontextabhängigkeit zu realisieren?")	<EINS>a	a am Zeilenanfang, wenn der Startzustand EINS die Bedeutung "am Zeilenanfang sein" hat

Syntax	Interpretation	Beispiel	
Ein regulärer Ausdruck ist:	Der linksstehende reguläre Ausdruck bezeichnet	regulärer Ausdruck:	passende Zeichenfolge:
r{m,n} r ein regulärer Ausdruck, m und n ganze Zahlen, für die m ⩽ n gelten muß	eine Folge von m, m+1,...,n passenden Zeichenreihen für r	a{1,4}	a, aa, aaa oder aaaa
{d} d ein Definitionsname (siehe unter "Format eines lex-Quell-Programms")	eine Zeichenreihe, die zu dem regulären Ausdruck paßt, der als d definiert ist	{zahl}	0,1,2,3,4,5, 6,7,8 oder 9, wenn die Definition lautet: zahl [0-9]

Aktionen in lex-Quell-Programmen

Wenn eine Textstelle im Eingabestrom zu einem regulären Ausdruck paßt, dann wird die Aktion ausgeführt, die zu dem entsprechenden regulären Ausdruck gehört. Textstellen, die nicht zu regulären Ausdrücken passen, werden kopiert.

Wenn lex mit yacc zusammenarbeiten soll, muß normalerweise der gesamte Eingabestrom zu regulären Ausdrücken passen und jede Textstelle muß eine Aktion veranlassen, die nicht nur aus Kopieren besteht.

Eine Aktion wird in C geschrieben. Sie kann aus einer einzelnen Anweisung oder aus Vereinbarungen und Anweisungen, die zu einem Block zusammengefaßt sind, bestehen. Anweisungen werden mit einem ";" abgeschlossen. Die geschweiften Klammern "{" und "}" dienen, wie in C üblich, dazu, Vereinbarungen und Anweisungen zu einem Block zusammenzufassen.

Spezielle Aktionen

; leere Anweisung (eine Textstelle, die zum dazugehörenden regulären Ausdruck paßt, wird ignoriert).

| Wiederholungsaktion, d.h. die Aktion soll dieselbe sein wie die Aktion, die zur nächsten Regel gehört.

Spezielle Variablen und Funktionen

Im Programm, das lex generiert, sind spezielle Variablen und Funktionen definiert, auf die auch die Aktionen oder Benutzer-Funktionen zugreifen können:

yytext	yytext ist die Adresse eines externen Zeichenfeldes (char *). In diesem Feld steht die aktuelle Zeichenreihe, die zu einem regulären Ausdruck paßt. Das Zeichenfeld hat die Länge YYLMAX. Die Konstante YYLMAX hat den Wert 200. Sie können sie undefinieren.
yyleng	yyleng gibt die Länge der gefundenen Zeichenreihe in yytext an. Eine aktuelle passende Zeichenreihe steht also in yytext[0], yytext[1],..., yytext[yyleng-1].
yymore()	Üblicherweise wird der Inhalt von yytext überschrieben, wenn die nächste Zeichenreihe gefunden wird, die zu einem regulären Ausdruck paßt. yymore veranlaßt, daß der Inhalt von yytext nicht überschrieben wird, und daß die nächste passende Zeichenreihe ans Ende von yytext angehängt wird. Mit yymore können Sie die Zeichenreihe verlängern, die eine bestimmte Aktion veranlassen soll.
yyless(n)	yyless bewirkt, daß nur die ersten *n* Zeichen der aktuellen passenden Zeichenreihe, die in yytext steht, eine bestimmte Aktion veranlassen. Restliche Zeichen, die noch in yytext stehen, werden an den Eingabestrom zurückgegeben und erneut als Eingabe behandelt.
input	input holt das nächste Eingabezeichen vom Eingabestrom yyin. yyin ist definiert als FILE *yyin = {stdin};.
output(c)	output schreibt das Zeichen *c* in den Ausgabestrom yyout. yyout ist definiert als FILE *yyout = {stdout};.
unput(c)	unput gibt das Zeichen *c* an den Eingabestrom zurück, damit *c* später wieder mit input eingelesen werden kann.

Die Ein-/Ausgabe-Funktionen sind als Makros definiert. Der Benutzer kann eigene Versionen für diese Funktionen bereitstellen. Die Funktionen müssen untereinander konsistent sein.

input betrachtet eine binäre Null als "Dateiende". "Dateiende" führt dazu, daß yylex beendet wird. Sie müssen input ändern, um auch Dateien bearbeiten zu können, bei denen die binäre Null eine andere Bedeutung hat.

Der return-Wert 0 bei input muß die Bedeutung "Dateiende" haben.

Die Funktionen unput und input müssen in Beziehung zueinander stehen, um ein Vorauslesen zu ermöglichen. Ein Vorauslesen ist nötig bei jedem regulären Ausdruck,

- der das Zeichen "/" enthält, oder
- der mit einem der Zeichen +, *, ? oder $ endet, oder
- der Präfix eines anderen regulären Ausdrucks ist.

Das Vorauslesen ist auf 100 Zeichen beschränkt.

yywrap() — yywrap wird jedes Mal aufgerufen, wenn im Eingabestrom ein Dateiende erreicht wird. Der return-Wert 1 von yywrap hat die Bedeutung "Ende des Eingabestroms". Es wird keine weitere Eingabedatei mehr behandelt. Um weitere Dateien eingeben zu können, muß der Benutzer eine eigene Version von yywrap zur Verfügung stellen. Diese sollte den return-Wert 0 haben, wenn weitere Eingabe folgt.

REJECT — REJECT ist eine Anweisung. Sie bewirkt, daß zu der Regel übergegangen wird, zu deren regulärem Ausdruck die aktuelle Zeichenreihe am nächstbesten paßt (siehe unter "Regeln bei lex-Quell-Programmen"). REJECT liest keine neuen Zeichen ein. Erst wenn keine zweite Alternative existiert, liest REJECT weiter im Eingabetext, vergißt aber nicht die aktuelle Zeichenreihe von vorher. REJECT versucht eine zweite Alternative zur erweiterten aktuellen Zeichenreihe zu finden.

REJECT hilft beim Erstellen von Statistiken. Sie können z.B. zählen, wie oft spezifizierte Zeichenreihen in einem Eingabestrom vorkommen. Mit REJECT können Sie auch Zeichenreihen zählen, die sich überlagern.

ECHO — Die Zeichenreihe, die in yytext steht, wird in den Ausgabestrom kopiert.

Beispiele für Aktionen

1. Beispiel für die Verwendung von "|":

```
%%
" "     |
"\t"    |
"\n"    ;
```

Diese drei Regeln können auch zu einer zusammengefaßt werden:

```
%%
[ \t\n] ;
```

Leerzeichen, Tabulatorzeichen und "Neue Zeile" im Eingabestrom werden ignoriert.

2. Beispiel für die Verwendung von yytext, yyleng und yyless(n):

Gegeben seien die Regeln:

```
%%
Donald\ Duck    {
                yyless(yyleng-5);
                printf("%s\n", yytext);
                }
Duck            ;
```

Wenn im Eingabestrom die Zeichenreihe

```
Donald Duck
```

vorkommt, so wird im Ausgabestrom

```
Donald
```

ausgegeben.

3. Beispiel für die Verwendung von REJECT:

   Gegeben seien die Regeln:

```
%%
sch    {printf("sch gefunden\n"); REJECT;}
ch     printf("ch gefunden\n");
\n     |
.      ;
```

   Jedesmal wenn in einem Eingabetext "ch" oder "sch" vorkommt, wird es ausgedruckt. Auch "ch" im "sch" soll ausgedruckt werden. Bei der Eingabe

```
acht schlechte schokoladen-weihnachtsmänner schmecken
nicht.
```

   wird dreimal

   sch gefunden

   und siebenmal

   ch gefunden

   ausgegeben.

4. Noch ein Beispiel zu REJECT:

```
%{
int diagram[128][128];
%}
%%
[a-z][a-z]    {diagram[yytext[0]][yytext[1]]++; REJECT;}
.             ;
\n            ;
```

   Es wird gezählt, wie oft Buchstabenpaare im Text vorkommen. Bei der Eingabe "hallo!", erhöhen sich

   diagram[h][a]
   diagram[a][l]
   diagram[l][l] und
   diagram[l][o].

Welche Regel wird verwendet?

Wenn eine Zeichenreihe zu mehr als einem regulären Ausdruck paßt, dann gilt:

- Es wird die längstmögliche passende Zeichenreihe ausgesucht und es wird die Aktion ausgeführt, die bei dem regulären Ausdruck steht, zu dem diese Zeichenreihe paßt.
- Paßt eine Zeichenreihe mit der gleichen Anzahl von Zeichen zu mehreren Regeln, so wird unter diesen Regeln die ausgewählt, die Sie im lex-Quell-Programm als erste angegeben haben.
- Nach REJECT wird die Aktion der Regel ausgeführt, zu der die Zeichenreihe am nächstbesten paßt. Gibt es keine zweite Alternative, wird der Eingabetext weiter eingelesen.

Beispiel

Die Regeln:

```
%%
heidi     printf("Hallo!");
[a-z]+    printf("Grüß Gott!");
```

bewirken, daß bei der Eingabe

```
heidi
```

auf der Ausgabe

```
Hallo!
```

ausgegeben wird. Die etwas veränderten Regeln

```
%%
heidi     {REJECT; printf("Hallo!");}
[a-z]+    printf("Grüß Gott!");
```

bewirken bei der gleichen Eingabe die Ausgabe

```
Grüß Gott!
```

Die Anweisung printf("Hallo!"); wird nie ausgeführt.

Welche Möglichkeiten gibt es Kontextabhängigkeit zu realisieren?

Die Operatorzeichen "/", "+", "*", "?" und "$" dienen dazu, die Anwendung von Regeln vom Rechtskontext abhängig zu machen. Ob eine Regel mit diesen Operatorzeichen angewendet wird, hängt davon ab, welche Zeichenfolgen unmittelbar nacheinander im Eingabestrom stehen. Durch Vorauslesen des Textes kann Rechtskontextabhängigkeit realisiert werden.

Einfache Linkskontextabhängigkeit, nämlich Zeilenanfang, wird mit dem Operatorzeichen "^" ausgedrückt.

Schwieriger ist das Problem zu lösen, wenn eine Aktion von einem Linkskontext abhängt, der nicht unmittelbar vorausging, sondern "einige Zeit früher" vorkam. Lösungsmöglichkeiten dafür sind:

- globale Variablen setzen.

  Sie belegen eine globale Variable mit einem Wert, der vom Eingabetext abhängt. Der Wert der Variablen legt fest, wie ein folgender Text zu behandeln ist und welche Aktionen auszuführen sind. Diese Lösungsmöglichkeit sollten Sie nur benutzen, wenn wenige Regeln vom Linkskontext abhängig sind.

- Regeln mit regulären Ausdrücken verwenden, in denen Startzustände vorkommen (siehe unter "Reguläre Ausdrücke in lex-Quell-Programmen").

  Sie können jede beliebige Regel von einem Startzustand abhängig machen und den aktuellen Startzustand jederzeit ändern. Die Größe eines Scanners verringert sich, wenn Sie Startzustände einführen.

- Mehrere Programme zur lexikalischen Analyse zusammen ablaufen lassen.

  Wenn die Regeln, die von einem bestimmten Vortext abhängen, voneinander sehr verschieden sind, ist es oft übersichtlicher, mehrere Programme zu schreiben. Je nach Vortext wird dann ein entsprechendes Programm aufgerufen.

  Bei dieser Methode ist darauf zu achten, daß mehrere Programme, die lex generiert hat, nicht direkt zu einem Gesamtprogramm zusammengebunden werden können. Es würden Namenskonflikte auftreten.

Beispiel

Ein Text wird unverändert kopiert. Nur das Wort "Name" soll ersetzt werden und zwar

- durch "Roland", wenn am Zeilenanfang der Buchstabe a stand,
- durch "Martin", wenn am Zeilenanfang der Buchstabe b stand,
- durch "Klaus", wenn am Zeilenanfang der Buchstabe c stand.

Mit globalen Variablen läßt sich das Problem folgendermaßen lösen:

```
        int signal;
%%

^a      {signal = 'a'; ECHO;}

^b      {signal = 'b'; ECHO;}

^c      {signal = 'c'; ECHO;}

\n      {signal =  0 ; ECHO;}

Name    {
        switch (signal)
               {
               case 'a': printf("Roland"); break;
               case 'b': printf("Martin"); break;
               case 'c': printf("Klaus"); break;
               default:  ECHO; break;
               }
         }
```

Die Lösungsmöglichkeit mit Startzuständen sieht so aus:

Sie müssen jeden Startzustand im lex-Quell-Programm definieren (siehe unter "Definitionen im lex-Quell-Programm"). Um in einen Startzustand *start* zu kommen, muß die Aktion

```
BEGIN     start;
```

ausgeführt werden. Die Aktion

```
BEGIN     0;
```

stellt den ursprünglichen Anfangszustand des Automaten wieder her.

Die Lösung lautet also:

```
%START AA BB CC

%%

^a          {ECHO; BEGIN AA;}

^b          {ECHO; BEGIN BB;}

^c          {ECHO; BEGIN CC;}

\n          {ECHO; BEGIN 0;}

<AA>Name    printf("Roland");
<BB>Name    printf("Martin");
<CC>Name    printf("Klaus");
```

Bei der zweiten Lösung übernimmt lex die Arbeit, die bei der ersten Lösung vom Benutzer-Code geleistet wird.

Der dritte Lösungsweg (mit mehreren Programmen) soll hier nicht gezeigt werden. Er ist nur sinnvoll bei komplexeren Aufgaben.

lex und yacc

In dem Programm, das lex erstellt, wird eine Funktion yylex definiert. Die Funktion, die die lexikalische Analyse bei yacc übernehmen soll (der Scanner), muß auch yylex heißen. lex und yacc passen also gut zusammen.

Normalerweise wird lex von einer Funktion main aufgerufen, die in der lex-Bibliothek steht. Wenn Sie yacc zusammen mit lex verwenden, dann erfolgt der Aufruf von yylex in dem Parser, den yacc generiert.

Wenn der Parser eine neue Eingabe benötigt, ruft er yylex auf. Der Scanner liest so viele Zeichen der Eingabe, bis er erkennt, ob ein Token und welches Token oder ob ein Eingabefehler vorliegt. Im ersten Fall wird eine Nummer an den Parser zurückgegeben, die eindeutig das gefundene Token bezeichnet (siehe yacc). Im zweiten Fall wird der Fehler zurückgemeldet.

Jede lex-Regel sollte (bei Zusammenarbeit von lex und yacc) mit der Aktion

```
return(TOKEN);
```

enden und in der Aktion einen passenden Wert für *TOKEN* zurückgeben.

Das C-Programm, das lex generiert, kann als Teil des Programms übersetzt werden, das yacc erstellt. Dies geschieht, indem Sie im letzten Abschnitt des yacc-Quell-Programms die Zeile

```
#include "lex.yy.c"
```

einfügen.

Wenn das yacc-Quell-Programm z.B. in der Datei *grammatik*, das lex-Quell-Programm in der Datei *regeln* und ein Eingabestrom in der Datei *eingabe* stehen, dann lautet die Kommandofolge:

```
$ yacc grammatik
$ lex regeln
$ cc y.tab.c -ly -lln
$ a.out < eingabe
```

Sie müssen die yacc-Bibliothek vor der lex-Bibliothek binden, um ein Hauptprogramm zu haben, das den yacc-Parser aufruft. Die Aufrufe von yacc und lex können Sie vertauschen.

Fehlerdiagnose

Fehlermeldungen können auftreten

- nach dem Aufruf von lex,
- beim Übersetzen und Binden von lex.yy.c,
- beim Ausführen des lex-Programms.

Für den Benutzer ist es oft schwer zu erkennen, wo ein Fehler liegt. lex gibt meist nur Fehler in den eigentlichen lex-Definitionen oder Regeln aus. Aktionen und Funktionen, die gegen C-Sprachregeln verstoßen, und fehlerhafte Textstellen, die lex kopiert, führen erst beim Übersetzen oder Ausführen zu Fehlermeldungen.

Version 2.0

Die Beschreibung von lex ist für alle Versionen (1.0B, 1.0C, 2.0) gültig.

Beispiele

1. Ein einfaches Beispiel:

   lex soll ein C-Programm erstellen, das

   - Großbuchstaben durch die entsprechenden Kleinbuchstaben ersetzt,
   - Leerzeichen am Zeilenende entfernt,
   - aufeinanderfolgende Leerzeichen innerhalb einer Zeile durch ein einziges Leerzeichen ersetzt.

   In der Datei *regeln* steht folgendes lex-Quell-Programm:

```
%%
[A-Z]   putchar(yytext[0] + 'a' - 'A');
[ ]+$   ;
[ ]+    putchar(' ');
```

   Das Zeichen"%%"zeigt den Anfang der Regeln an. Es sind drei Regeln vorgegeben.

   Die erste Regel

```
[A-Z] putchar(yytext[0] + 'a' - 'A');
```

   enthält

   - den regulären Ausdruck "[A-Z]",
   - die Aktion "putchar(yytext[0] + 'a' - 'A');".

   Der reguläre Ausdruck "[A-Z]" bezeichnet die Menge aller Großbuchstaben.
   Wenn in einem Eingabestrom ein Großbuchstabe vorkommt, dann soll statt diesem der entsprechende Kleinbuchstabe im Ausgabestrom ausgegeben werden. Ein C-Ausdruck, der diese Aktion veranlaßt, ist

```
putchar(yytext[0] + 'a' - 'A');
```

   In yytext steht die Zeichenfolge des Eingabestroms, die zu dem regulären Ausdruck paßt, hier also ein Großbuchstabe. yytext[0] bezeichnet das erste (und hier auch einzige) Element des Feldes yytext. Die Umwandlung eines Großbuchstaben in den entsprechenden Kleinbuchstaben erfolgt durch Addition des Abstands zwischen Groß- und Kleinbuchstaben, "'a' - 'A'". putchar fügt dem Ausgabestrom, der auf der Standard-Ausgabe erstellt wird, den Kleinbuchstaben an.

Die nächsten beiden Regeln

```
[ ]+$  ;
[ ]+   putchar(' ');
```

behandeln das Auftreten von mindestens einem Leerzeichen. Der reguläre Ausdruck dafür lautet "[ ]+". "[ ]" bezeichnet ein Leerzeichen, "+" bedeutet "mindestens eines". "$" bezeichnet das Zeilenende.

Treten in einem Eingabestrom Leerzeichen nicht am Zeilenende auf, so passen sie nur zum regulären Ausdruck der letzten Regel. In diesem Fall wird statt einem oder mehreren aufeinanderfolgenden ein einziges Leerzeichen dem Ausgabestrom auf der Standard-Ausgabe zugefügt. Die Aktion dafür lautet

```
putchar(' ');
```

Leerzeichen am Zeilenende passen zu [ ]+$. Die Aktion dazu besteht nur aus der leeren Anweisung ";". Diese Leerzeichen werden also ignoriert und nicht kopiert.

Sämtliche Zeichen des Eingabestroms, die nicht zu den regulären Ausdrücken der drei Regeln passen, werden auf die Standard-Ausgabe kopiert.

lex wird aufgerufen und das lex-Programm übersetzt und gebunden:

```
$
$
```

Mit dem übersetzten Programm wird der folgende Text in der Datei *text* behandelt:

```
Eine     kleine    Micky Maus
zieht     sich    ihre Hose aus,
zieht sie wieder     an
und Du bist    dran.
```

```
$
```

In der Datei *textneu* steht jetzt folgender Text:

```
eine kleine micky maus
zieht sich ihre hose aus,
zieht sie wieder an
und du bist dran.
```

2. Beispiel:

   Das C-Programm, das lex aus dem folgenden lex-Quell-Code erstellt, gibt eine Statistik aus. Es zählt, wie oft im Eingabestrom Ziffern, Buchstaben oder sonstige Zeichen vorkommen. Die Zeichen "Dateiende" und "Neue Zeile" sollen nicht mitgezählt werden. Der lex-Quell-Code dazu lautet:

```
 /* Definitionen */
%{
int bu = 0, zi = 0, so = 0;
%}
Zi [0-9]
Bu [a-zA-Z]
So [^0-9a-zA-Z\4\n]

% %
 /* Regeln */

\n      ;

{Bu}    bu++;

{Zi}    zi++;

{So}    so++;

% %
/* Benutzer-Funktion */
yywrap()
{
    printf("Buchstaben: %d\n",bu);
    printf("Ziffern: %d\n",zi);
    printf("Sonstige Zeichen: %d\n",so);
    return(1);
}
```

Dateien

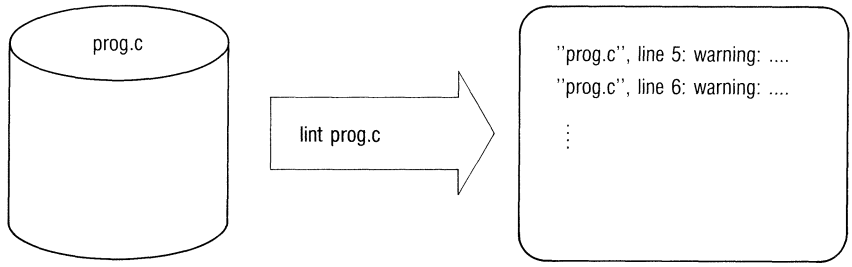
lex.yy.c	enthält das C-Programm, das lex erstellt.
/lib/libln.a	lex-Bibliothek

Siehe auch

M.E.Lesk und E.Schmidt,
LEX - Lexical Analyzer Generator

> > > > yacc

C-Programme überprüfen - a C program verifier



lint durchsucht C-Quell-Programme

- nach möglichen (syntaktischen und semantischen) Fehlern,
- nach nicht portierbaren Programmabschnitten,
- nach Möglichkeiten, Rechenzeit und Speicher zu sparen.

lint prüft verschiedene C-Sprachregeln strenger als der C-Übersetzer, der C-Programme möglichst schnell und effizient in ausführbare Dateien übersetzt.

Auf der Standard-Ausgabe gibt lint aus, was ihm an einem C-Programm nicht gefällt.

lint[␣-schalter...]␣datei...

schalter

kein Schalter

lint versucht zu erkennen, ob

– Funktionsaufrufe verträglich sind mit den jeweiligen Funktionsdefinitionen,

- die Datentypen und die Anzahl der Argumente bei Funktionsaufrufen mit der jeweiligen Definition übereinstimmen,

- return-Werte niemals, nur manchmal, oder falsch verwendet werden,
- Variablen oder Funktionen definiert, aber niemals verwendet werden (Ausnahme: explizit extern deklarierte Variablen),
- lokale Variablen initialisiert, aber niemals verwendet werden,
- lokale Variablen verwendet werden, bevor ihnen ein Wert zugewiesen wird,

- Datentypen bei Zeigern übereinstimmen,
- die Namen der Konstanten einer Aufzählung und der Name der Aufzählung voneinander und von allen anderen Namen verschieden sind,
- bei Aufzählungen keine unerlaubten Operationen auftreten,
- bei Strukturen und Varianten die Datentypen beim "->"-Operator richtig verwendet werden,

- veraltete C-Syntax verwendet wird,

- eine Programmstelle niemals durchlaufen wird (Ausnahme: lint ignoriert unerreichbare break-Anweisungen),
- eine Programmschleife niemals durchlaufen wird oder niemals verlassen werden kann (z.B. "while (1)"),
- Programmschleifen nicht von Anfang an durchlaufen werden,
- direkt auf break-, continue-, goto- oder return-Anweisungen unerreichbare Programmstellen folgen,

- das C-Quell-Programm nicht portierbare Zuweisungen oder Vergleiche enthält,
- bei Bit-Manipulationen keine unsigned Werte oder unsigned Variablen verwendet werden,

– die Reihenfolge der Auswertung von Ausdrücken maschinenabhängig ist (z.B. bei "a[i] = b[i + +]"),

– ein (syntaktisch und semantisch) richtiger Programmabschnitt zu Laufzeitfehlern führen könnte,
– nicht effizienter Programmierstil vorliegt.

a — lint berichtet, ob Zuweisungen von long Werten an int oder short Variablen erfolgen. Aus solchen Zuweisungen können sich Fehler ergeben, wenn in einem Programm die Datentypen mit typedef unvollständig geändert werden.

b — lint zählt break-Anweisungen auf, die an Programmstellen stehen, die niemals durchlaufen werden.

Es muß nicht unbedingt ein Programmierfehler sein, wenn ein Programm solche Stellen enthält (lex- und yacc-Programme enthalten z.B. solche break-Anweisungen).

c — lint beklagt sich über cast-Anweisungen mit fragwürdiger Portierbarkeit.

h — lint versucht

– Laufzeitfehler intuitiv (was auch immer das heißen mag) zu erkennen,
– mögliche Stilverbesserungen anzugeben,
– herauszufinden, ob und wo Rechenzeit und Speicherplatz gespart werden können.

lxyz — lint durchsucht die Datei /usr/lib/llib-*lxyz* nach Funktionsdefinitionen. lint prüft dabei die Verträglichkeit des C-Quell-Programms mit den Funktionsdefinitionen in dieser Datei.

Beispiel

Die Datei /usr/lib/llib-lm enthält Dummy-Funktionen der mathematischen Funktionen, die in der Bibliothek /lib/libm.a zusammengefaßt sind. Eine Dummy-Funktion besteht aus dem entsprechenden Funktionskopf und der return-Anweisung. lint benötigt nur diese Angaben, um die Verträglichkeit des C-Quell-Programms mit den mathematischen Funktionen zu prüfen.

n — lint prüft nicht die Verträglichkeit der genannten Dateien mit der Standard-Bibliothek.

p — lint versucht herauszufinden, ob die genannten Dateien auf andere C-Dialekte (wie z.B. IBM und GCOS) portierbar sind. /usr/lib/llib-port enthält Dummy-Funktionen der portierbaren Funktionen.

s — wie Schalter h.

u — lint unterläßt Meldungen über Funktionen und Variablen, die

- benutzt, aber nicht definiert werden,
- definiert, aber nicht benutzt werden.

Dieser Schalter ist angebracht, wenn mit lint nur ein Teil der Dateien eines großen Programmsystems überprüft wird.

v — lint unterläßt Meldungen über Argumente in Funktionen, wenn diese Argumente nie benutzt werden. lint bemängelt aber register-Argumente, die nicht benutzt werden.

x — lint zählt die Variablen auf, die extern deklariert, aber nie benutzt werden.

schalter des cc-Kommandos:

Einige Schalter von cc können Sie auch bei lint verwenden:

-**D**name[=def]
name wird definiert, wie bei einer #define-Anweisung in einem C-Programm. Fehlt die Definition *def*, so wird *name* als 1 definiert.

-**U**name — Definitionen, bei denen *name* initialisiert wurde, werden entfernt.

-Idir
: lint sucht nach Include-Dateien, deren Name nicht mit "/" beginnt und deren Name von "<" und ">" eingeschlossen wird,

  – zuerst im Dateiverzeichnis der Dateien, die Sie als *datei* angeben,

  – dann in den Dateiverzeichnissen *dir* (Sie können den Schalter -I mehrmals angeben),

  – dann im Dateiverzeichnis /usr/include.

datei
: *datei* ist ein Dateiname. lint überprüft die Datei mit diesem Namen. In der Datei sollte ein C-Quell-Programm stehen.

  lint geht davon aus, daß alle angegebenen Dateien zusammen gebunden werden sollen. lint untersucht deshalb, ob die Dateien untereinander und mit der Standard-Bibliothek kompatibel sind. lint bemängelt z.B., wenn Standard-Funktionen im C-Programm nochmals definiert werden.

Anweisungs-Kommentare

Durch Kommentare im C-Quell-Programm ist es möglich, das Verhalten von lint zu beeinflussen. Diese Kommentare werden von lint beachtet, nicht aber vom Übersetzer. Solche Kommentare sind vor allem dann angebracht, wenn Sie erkennen, daß lint Programmstellen bemängeln würde, die keine Fehler sind und die durchaus vernünftigen Programmierstil darstellen.

/*ARGSUSED*/
: Für die nächste Funktion wird der Schalter v eingeschaltet.

/*LINTLIBRARY*/

Wenn dieser Kommentar am Dateianfang steht, dann betrachtet lint die Datei als eine Bibliotheksdatei, in der Funktionen definiert werden. Die Funktionen sind sog. Dummy-Funktionen, die kaum Anweisungen oder Aktionen enthalten, sondern nur

- den entsprechenden Funktionskopf (mit der Deklaration des return-Typs der Funktion, dem Funktionsnamen, den Funktionsargumenten und der Deklaration der Argumente)
- und return-Anweisungen.

lint verzichtet darauf, die Funktionen der lintlibrary-Datei aufzulisten, die nicht aufgerufen werden.

lint testet das C-Quell-Programm auf Verträglichkeit mit den Funktionen der lintlibrary-Datei hinsichtlich return-Anweisungen, dem return-Typ der Funktionen, Anzahl und Datentypen der Funktionsargumente.

Wenn Funktionen nicht in lintlibrary-Dateien stehen, testet lint auf Verträglichkeit mit /usr/lib/llib-lc oder /usr/lib/llib-port (Schalter p).

/*NOSTRICT*/

lint unterläßt es, genauer zu untersuchen, ob die Datentypen beim nächsten Ausdruck im C-Programm richtig verwendet werden.

/*NOTREACHED*/

Steht dieser Kommentar an einer Programmstelle, die niemals durchlaufen wird, so gibt lint keine Meldung darüber aus.

/***VARARGS**[n]*/

Steht dieser Kommentar vor einer Funktionsdefinition, prüft lint nicht, ob die Anzahl der Argumente bei verschiedenen Aufrufen dieser Funktion immer gleich ist.

Ist *n* eine natürliche Zahl (>0), dann überprüft lint die Datentypen der ersten *n* Argumente. Fehlt *n*, so wird der Datentyp keines Arguments untersucht.

Hinweis

- Es ist unmöglich, daß lint alle nötigen Angaben zu einem C-Programm bekommt, um Fehleranalysen zu erstellen, die immer korrekt sind.

  Sie sollten daher beachten:

  — Kommentare, die lint ausgibt, müssen nicht immer wahr sein.

  — Nicht alles, was lint bemängelt, muß ein Fehler oder muß schlechter Programmierstil sein.

- lint kann das Verhalten von Funktionen nicht verstehen, die keine return-Anweisung enthalten (z.B. exit). Die Kommentare von lint können in einem solchen Fall falsch sein.

- lint geht davon aus, daß eine Funktion, die nie erwähnt wird, auch nie aufgerufen werden kann. Umgekehrt, daß eine Funktion, die erwähnt wird, auch aufgerufen werden kann. Diese Annahmen müssen nicht immer richtig sein, treffen aber in den meisten Fällen zu.

Version 2.0

Version 2.0 kennt kein Kommando lint. cc übernimmt die Aufgabe von lint und gibt detailliertere Fehlerangaben aus als bei Version 1.0B/1.0C.

Beispiele

1. Fehlersuche bei mehreren Dateien:

   $

   datei1 und *datei2* enthalten C-Quell-Programme, die zusammen übersetzt und gebunden werden sollen. lint untersucht beide Programme auf Inkonsistenz und auf Effizienz.

2. Portabilität prüfen:

   $

   lint prüft zusätzlich zu 1. noch die Portabilität der Programme auf andere Betriebssysteme und andere Rechenanlagen.

3. Erkennen von eigenartigen Programmkonstruktionen:

   In der Datei *lampedusa.c* steht folgendes C-Programm:

```
main()
{
     unsigned x;
     x = 1;
     if (x<0) continue;
     if (1 != 0) x = 2;
     x = x<<2 + 40;
 }
```

```
$ lint lampedusa.c
```

   lint gibt folgende Meldungen aus:

```
"lampedusa.c", line 5: warning: unsigned comparison with 0?
"lampedusa.c", line 5: illegal continue
$
```

```
$ lint -h lampedusa.c
```

   lint gibt zusätzlich folgende Meldungen aus:

```
"lampedusa.c", line 6: warning: constant in conditional context
"lampedusa.c", line 6: warning: constant in conditional context
```

4. Erkennen von cast-Anweisungen mit fragwürdiger Portierbarkeit:

   In der Datei *kreta.c* steht folgendes C-Programm:

```
main()
{
    char *p;
    p = 1;
}
```

```
$ lint kreta.c
```

   lint gibt folgende Meldungen aus:

```
"kreta.c", line 4: warning: illegal combination of pointer and integer
"kreta.c", line 4: warning: p set but not used in function main
$
```

Zeile 4 wird geändert zu:

```
    p = (char *)1;
```

```
$ lint kreta.c
"kreta.c", line 4: warning: p set but not used in function main
$
```

Die erste Warnung unterbleibt.

```
$ lint -c kreta.c
"kreta.c", line 4: warning: illegal combination of pointer and integer
"kreta.c", line 4: warning: p set but not used in function main
$
```

lint gibt wieder beide Warnungen aus, denn Zeile 4 enthält eine cast-Anweisung mit fragwürdiger Portierbarkeit.

5. Veraltete C-Syntax erkennen:

   In der Datei *syros.c* steht folgendes C-Programm:

```
main()
{
int x (-1);             /* heute: int x = (-1); */
x = 2;
}
```

```
$ lint syros.c
```

lint gibt folgende Meldungen aus:

```
"syros.c", line 3: syntax error
"syros.c", line 3: warning: old-fashioned inizialisation: use =
"syros.c", line 3: warning: illegal combination of pointer and integer
"syros.c", line 3: unknown size
"syros.c", line 3: cannot recover from earlier errors: goodbye!
$
```

lint versteht also überhaupt nichts. Wird die 3. Zeile geändert zu:

```
int x = (-1);          oder          int x = -1;
```

so gibt lint keine Meldungen mehr zu Zeile 3 aus.

Ein weiteres Beispiel für veraltete C-Syntax:

Früher konnte geschrieben werden:

```
a =-1;                 für           a = a-1;
```

heute ist es unklar, was gemeint ist:

```
a -= 1;                oder          a = -1; ?
```

Dateien

/usr/lib/lint[12]
: enthält Programme.

/usr/lib/llib-lc
: enthält die Dummy-Funktionen der Standard-Funktionen.

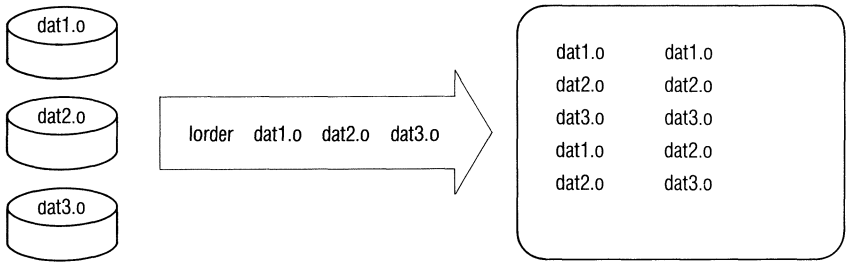
/usr/lib/llib-port
: enthält die Dummy-Funktionen der portierbaren Funktionen.

Siehe auch

S.C.Johnson,
Lint, a C Program Checker

> > > > cc

Objektmodule ordnen - find ordering relation for an object library



lorder ordnet Objektmodule nach externen Referenzen.

lorder erwartet als Eingabe einen oder mehrere Objektmodule. Die Objektmodule können auch in einer Bibliothek stehen.

Auf die Standard-Ausgabe schreibt lorder eine Liste. Jeder Listeneintrag besteht aus einem Paar von Objektmodulnamen. Der erste der Module, deren Namen zu einem Paar zusammengefaßt sind, enthält externe Variablen oder Aufrufe von Funktionen, die im zweiten Modul definiert werden.

Sie können diese Liste an tsort übergeben. tsort versucht dann die partiell geordneten Dateien topologisch zu sortieren und (total) zu ordnen. Damit wird eine Referenzordnung der Objektmodule aufgestellt. ld kann in einem Durchlauf auf eine Bibliothek zugreifen, die so geordnet wurde.

lorder␣datei...

datei — *datei* kann sein

- ein Objektmodulname, der mit ".o" enden muß,
- der Name einer Bibliothek.

lorder behandelt alle angegebenen Objektmodule und alle Objektmodule, die in einer der angegebenen Bibliotheken enthalten sind.

Hinweis

- Objektmodulnamen müssen immer mit ".o" enden, auch wenn sie in einer Bibliothek stehen. lorder liefert bei anderen Endungen unverständliche Ergebnisse.
- lorder ist eine Shell-Prozedur, die u.a. die Kommandos nm, sed, sort und join verwendet.

Version 2.0

Die Beschreibung von lorder ist für alle Versionen (1.0B, 1.0C, 2.0) gültig.

Beispiele

1. Dateien ordnen:

   Inhalt von *datei1.c*:

```
#define ENDE 20
extern ausgabe();       /* Hauptprogramm ruft ausgabe(i)
                              aus datei2.c auf */
main()
{
        int i;
        for(i=0;i<ENDE;i++)
                        ausgabe(i);
}
```

   Inhalt von *datei2.c*:

```
                        /* Ausgabe von i und i*i */
extern quadrat();       /* Ausgabeprogramm ruft quadrat()
                                 aus datei3.c auf */
ausgabe(i)
int i;
{
        printf("Das Quadrat von %d lautet %d\n",i,quadrat(i));
}
```

Inhalt von *datei3.c*:

```
                                /* Berechnung von i*i */
        int quadrat(i)
        int i;
        {
                return(i*i);
        }
```

```
$
datei1:
datei2:
datei3:
$
datei1.o datei1.o
datei2.o datei2.o
datei3.o datei3.o
datei1.o datei2.o
datei2.o datei3.o
$
```

In *datei1* wird die Funktion *ausgabe* aufgerufen, deren Definition in *datei2* steht. In *datei2* wird eine Funktion aus *datei3* aufgerufen.

2. Geordnete Bibliothek erstellen:

```
$
$
q - datei1.o
q - datei2.o
q - datei3.o
$
```

lorder bildet Paare aus allen Dateien, die mit ".o" enden.
tsort sortiert die Dateien. ar erstellt eine Bibliothek *archiv.a*, in der die Dateien in dieser Reihenfolge stehen.

Dateien

*symref, *symdef
Zwischendateien

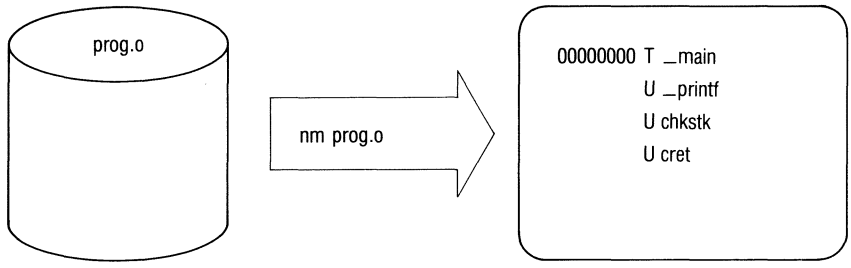
> > > > ar, join, ld, nm, sed, sort, tsort

Gruppen von Dateien verwalten

make dient zur Pflege einer Gruppe von Dateien, die voneinander abhängen. Das sind im allgemeinen Programme eines größeren Programmsystems, können aber auch beliebige Dateien sein. In einer Kommandodatei ("makefile") legen Sie fest, welche Kommandos ablaufen sollen, wenn sich bestimmte Dateien ändern.

Eine genaue Beschreibung von make finden Sie im Buch 1, Betriebssystem SINIX, Kapitel 6.

Symboltabelle ausgeben - print name list



nm gibt die Symboltabelle eines ablauffähigen Programms oder eines Objektmoduls auf der Standard-Ausgabe aus.

Jeder Tabelleneintrag besteht aus folgenden Angaben:

- dem Symbolwert (hexadezimal), der auch aus Leerzeichen bestehen kann (bei U). Der Symbolwert ist die relative Adresse des Symbols.
- der Art des Symbols, die durch einen der folgenden Buchstaben ausgedrückt wird:

  U undefiniertes Symbol,
  A absolutes Symbol,
  T Textsegmentsymbol,
  D Datensegmentsymbol,
  B bss-Segmentsymbol,
  F Dateinamensymbol,
  R Registername,
  C common-Symbol,

  (bei lokalen Symbolen wird der entsprechende Buchstabe klein, bei globalen groß geschrieben),
- dem Symbolnamen.

Die Tabelle ist alphabetisch nach den Symbolnamen sortiert.

nm[␣-schalter...][␣datei...]

schalter

g	nm listet nur die globalen Symbole auf.
n	nm sortiert numerisch nach Symbolwerten statt alphabetisch nach Symbolnamen.
o	Jeder Ausgabezeile wird der Datei- bzw. Bibliotheksname vorangestellt. Ohne diesen Schalter gibt nm den Namen nur einmal (am Anfang der Datei) an.
p	nm sortiert überhaupt nicht, sondern gibt die Symbole in der Reihenfolge aus, in der sie in der Symboltabelle auftreten.
r	Die Tabelle wird rückwärts geordnet ausgegeben.
u	nm listet nur Symbole auf, die undefiniert sind. Sie bezeichnen Referenzen, die noch nicht aufgelöst sind.
c	nm gibt nur die Symbole aus, die in einem C-Programm vorkommen. Die Namen dieser Symbole beginnen mit "_", wenn Sie den Schalter c nicht angeben.

datei

Sie geben als *datei* entweder den Namen eines Objektmoduls, eines ablauffähigen Programms oder einer Bibliothek an. nm gibt dann auf der Standard-Ausgabe die Symboltabelle von den Objektmodulen und Programmen aus, deren Name angegeben ist oder die in einer der angegebenen Bibliotheken stehen.

Standard (keine Angabe):
a.out

Version 2.0

Bei Version 2.0 hat ein Symbol der Art U einen Wert. Der Wert ist ein Index, der in eine Tabelle verweist. Diese Tabelle enthält Informationen über externe Referenzen, die noch nicht aufgelöst sind.

Das Kommando nm der Version 2.0 entspricht sonst dem Kommando nm der Version 1.0C.

Beispiel

In der Datei *programm.c* steht folgendes C-Programm:

```
main()
{
        int i=20;
        printf("%d",i*i);
}
```

```
$
$
```

Auf der Standard-Ausgabe wird aufgelistet (Version 1.0C):

```
00000000 T _main
         U _printf
         U chkstk
         U cret
$

$
```

Jetzt erscheint folgende Ausgabe:

```
00000000 T main
         U printf
$
```

Bei Version 1.0B lautet die erste Zeile der Ausgabe:

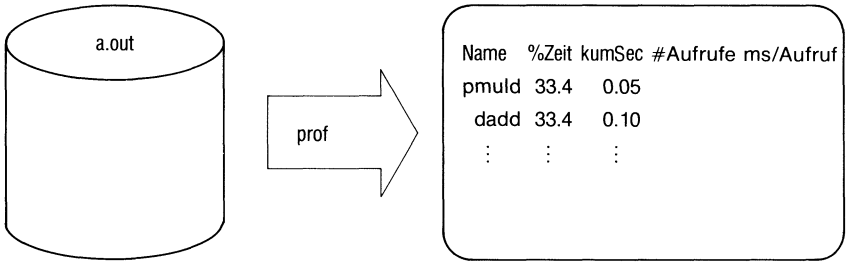
`0000 T _main` bzw. `0000 T main`

Dateien

a.out	nm gibt die Symboltabelle von a.out aus, wenn ein Argument *datei* fehlt.

> > > > ar

Zeittabelle eines Programms aufstellen - display profile data



prof stellt eine Zeittabelle für ein C-Programm auf. Die Tabelle zeigt auf, wie das Programm seine CPU-Zeit verbraucht. Sie erfahren z.B. wie oft eine Funktion aufgerufen wird und wieviel CPU-Zeit sie verbraucht.

prof wertet die Datei mon.out aus, um die Tabelle zu erstellen. mon.out wird von der Funktion monitor standardmäßig angelegt, wenn

- Sie ein C-Quell-Programm mit dem Schalter -p übersetzt haben und
- das Programm fehlerfrei abgelaufen ist.

prof bezieht die Daten, die in mon.out stehen, auf die Symboltabelle des übersetzten C-Programms.

prof gibt auf der Standard-Ausgabe für jedes nicht lokale Symbol (z.B. eine Funktion) eine Zeile mit folgenden Informationen aus:

- dem Symbolnamen (Spalte "name"),
- der gesamten Zeit (in Prozent), die das Symbol bei der Programmausführung verbraucht hat (Spalte "%time"),
- der Anzahl der Sekunden, die das Symbol und alle Symbole, die vorher aufgelistet wurden, verbraucht haben (Spalte "cumsecs"). Wieviel Zeit (in Sekunden) ein Symbol verbraucht hat, erfahren Sie, wenn Sie die Differenz bilden zwischen dieser Angabe und der Angabe aus der Zeile davor.
- der Anzahl der Aufrufe des Symbols (Spalte "# call"),

- den Millisekunden, die pro Aufruf verbraucht wurden (Spalte "ms/call").

Alle Zeiten sind CPU-Zeiten.

Die Namen der Spalten beziehen sich auf Version 1.0B. Bei Version 1.0C heißen die Spalten "Name", "%Zeit", "kum Sek", "# Aufrufe" und "ms/Aufruf".

Die Reihenfolge, in der die Symbole aufgelistet werden, ist so gewählt, daß die Ausführungszeit in Prozent abnimmt.

prof[␣-schalter...][␣datei]

schalter

a — prof wertet die Daten zu allen Symbolen, nicht nur zu den globalen, aus.

l — prof sortiert nach dem Symbolwert (d.h. der relativen Adresse), anstatt nach abnehmenden Prozentwerten.

datei — *datei* ist der Name eines Objektmoduls oder eines ausführbaren Programms. prof nimmt die Symboltabelle von *datei* um mon.out auszuwerten. Die Symboltabelle darf daher nicht fehlen.

datei muß von cc mit dem Schalter -p erstellt worden sein (siehe auch ld , "Hinweise"). Die Datei mon.out wird dann automatisch erstellt, wenn Sie das Programm in *datei* ablaufen lassen.

Standard (keine Angabe):
a.out.

Hinweis

Es können Quantelungsfehler auftreten. Das sind Fehler, die beim Auf- oder Abrunden von reellen Zahlen entstehen.
Z. B. ergeben sich bei Funktionen, die nur kurz laufen, nicht repräsentative Daten, wie Ausführungszeiten von 0.00 Sekunden. Die Auflösung beträgt 1/20 Sekunde.

Version 2.0

Das Kommando prof der Version 2.0 verhält sich wie das Kommando prof der Version 1.0C.

Beispiel

In der Datei *programm.c* steht folgendes C-Programm:

```
#include <math.h>          /* Primzahlengenerator */
#define ENDE 20

main()
{
     int    i,j;
     for(i=3;i<ENDE;i+=2){
           for(j=3;j<sqrt((double)i) && i%j;j+=2)
                 ;
           if (i%j != 0)
                 printf("%d ",i);
      }
      printf("\n");
}

$ cc -p programm.c -lm -lffp
$ a.out
5 7 11 13 17 19
$ prof  more
```

prof schreibt auf die Standard-Ausgabe (Version 1.0B):

```
    name %time  cumsecs  #call  ms/call
    ddiv  66.9     0.10
   pmuld  33.4     0.15
  _frexp   0.0     0.15
    ftol   0.0     0.15
    ltof   0.0     0.15
 _printf   0.0     0.15
  ...
   _main   0.0     0.15      1     0.00
  ...
$
```

Bei Version 1.0C lautet die Ausgabe:

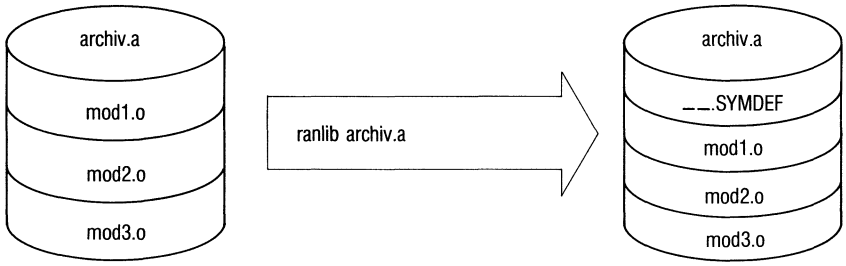
```
    Name %Zeit  kum Sek  #Aufrufe  ms/Aufruf
   pmuld  33.4     0.05
    dadd  33.4     0.10
    fldd  33.4     0.15
    ftol   0.0     0.15
    ltof   0.0     0.15
 _printf   0.0     0.15
  ...
   _main   0.0     0.15         1       0.00
  ...
$
```

Dateien

mon.out	enthält die Daten, die prof auswertet.
a.out	enthält die Symboltabelle, wenn *datei* fehlt.

> > > > monitor, profil, cc

Bibliothek mit einem Inhaltsverzeichnis versehen - convert archives to random libraries



ranlib legt ein Inhaltsverzeichnis für eine Bibliothek an. Das Inhaltsverzeichnis wird an den Anfang der Bibliothek gestellt. Es heißt "_ _.SYMDEF". Im Inhaltsverzeichnis stehen die Namen der Objektmodule, die in der Bibliothek enthalten sind. Um die Bibliothek dann wieder zu erstellen, ruft ranlib das Kommando ar auf. Im Dateisystem des aktuellen Dateiverzeichnisses muß deswegen genügend freier Platz für Zwischendateien vorhanden sein.

Sie müssen ranlib erneut aufrufen und "_ _.SYMDEF" aktualisieren, wenn Sie nachträglich einen Modul in die Bibliothek aufnehmen.

Bibliotheken mit Inhaltsverzeichnis kann ld schneller bearbeiten und binden.

ranlib archiv...

archiv — Die Datei *archiv* sollte eine Bibliothek sein, die Sie mit ar angelegt haben. ranlib nimmt nur die Namen von Bibliotheks-Dateien ins Inhaltsverzeichnis auf, die Objektmodule sind.

Hinweis

- Es können Laufzeitinkonsistenzen (Phasenfehler) auftreten, denn das Erstellen einer Bibliothek mit ar und das Anlegen eines Inhaltsverzeichnisses mit ranlib laufen getrennt ab.
- ld gibt eine Warnung aus, wenn nach dem Anlegen des Inhaltsverzeichnisses durch ranlib, an der Bibliothek etwas verändert wurde. Eine Warnung wird z.B. ausgegeben, wenn die Bibliothek kopiert wurde. Nach der Warnung durchsucht ld die Bibliothek dann nur sequentiell einmal.

Version 2.0

Die Beschreibung von ranlib ist für alle Versionen (1.0B, 1.0C, 2.0) gültig.

Bei Version 2.0 müssen alle Bibliotheken, die Sie mit einer Objektdatei zusammenbinden wollen (mit cc oder ld), ein Inhaltsverzeichnis enthalten. Sie behandeln am besten alle Bibliotheken, die Sie mit ar bearbeiten, sofort danach mit ranlib. Ein "makefile" (siehe Kommando "make", Betriebssystem SINIX, Buch 1) kann automatisch den ranlib-Aufruf übernehmen.

Beispiel

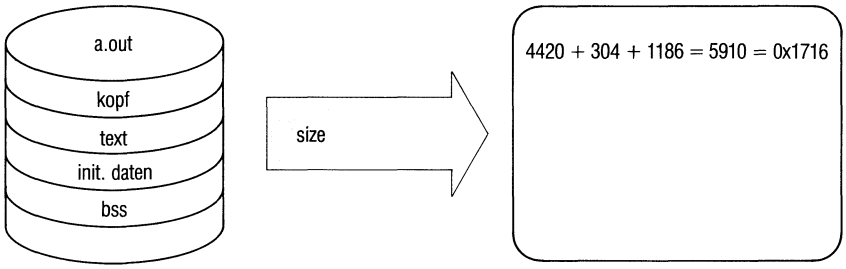
Sie haben die Bibliothek *archiv.a* noch nicht mit ranlib behandelt.

```
$
-rw-rw-r-- 1 gudrun     2970 Okt  1 13:52 archiv.a
$
$
-rw-rw-r-- 1 gudrun     3104 Okt  1 14:23 archiv.a
```

Die Bibliothek hat ein Inhaltsverzeichnis bekommen und ist deshalb grösser geworden. Bei Version 1.0C hat die Bibliothek die Standard-Schutzbiteinstellung `-rw-------` `statt` `-rw-rw-r--` (Version 1.0B).

> > > > ar, ld, copy, settime

Größe einer Objektdatei ausgeben - size of an object file



size gibt die Größe einer Objektdatei, also eines Objektmoduls oder eines ablauffähigen Programms, aus.

size schreibt auf die Standard-Ausgabe drei Zahlen (dezimal) und deren Summe. Bei Version 1.0C wird die Summe dezimal und hexadezimal, bei Version 1.0B dezimal und oktal ausgegeben.
Die drei Zahlen geben jeweils die Anzahl der Bytes an, die belegt werden

- vom Textsegment,
- vom Datensegment und
- vom bss-Segment.

Das Datensegment ist der initialisierte Datenbereich, das bss-Segment der nicht-initialisierte Datenbereich.

size[␣objekt...]

objekt — Die Datei *objekt* sollte eine Objektdatei sein.

Standard (keine Angabe):
a.out.

Version 2.0

Das Kommando size der Version 2.0 verhält sich wie das Kommando der Version 1.0C.

Beispiel

In der Datei *programm.c* steht folgendes C-Programm:

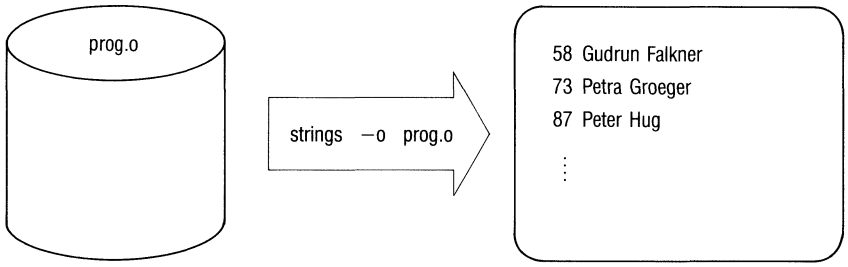
```
/* Beispielprogramm zum Gebrauch von <size> */

main()
{
      printf("Dies ist ein Beispielprogramm zur Bestimmung\n");
      printf("der Groesse von Objektfiles mit Hilfe\n");
      printf("des Befehls 'size'.\n");
}
```

```
$ cc programm.c
$ size
```

size schreibt auf die Standard-Ausgabe:

```
4420 + 304 + 1186 = 5910 = 0x1716
$
(Version 1.0C)

4420+304+1186 = 5910Db = 013426b
$
(Version 1.0B)
```

ASCII-Zeichenreihen in Binärdatei suchen - find the printable strings in an object, or other binary, file



strings sucht in Dateien mit beliebigem Inhalt nach ASCII-Zeichenreihen,

- die aus mindestens vier abdruckbaren Zeichen bestehen und
- die mit einem Nullbyte oder einer neuen Zeile enden.

In Objektdateien (Objektmodule oder ausführbare Programme) werden nur die initialisierten Daten untersucht (siehe aber Schalter -a).

strings kann bei vielen Problemen hilfreich sein. Mit strings können Sie z.B.

- unbekannte Objektmodule identifizieren oder
- in Objektmodulen prüfen, welche Funktions-Aufrufe erfolgen.

strings[␣schalter...][␣-z]␣datei...

schalter

-a	strings durchsucht die gesamte Binärdatei nach ASCII-Zeichenreihen.
–	Dieser Schalter bewirkt dasselbe wie -a.
-o	strings gibt vor jeder Zeichenreihe den Abstand (dezimal) zum Anfang der Datei an.

-z — Es werden nur Zeichenreihen berücksichtigt mit mindestens *z* druckbaren Zeichen.

Standard (keine Angabe):
z = 4.

datei — Die Datei *datei* sollte eine Binärdatei sein, also ein Objektmodul oder ein ausführbares Programm.

Hinweis

strings verwendet zum Identifizieren von Zeichenreihen einen sehr primitiven Algorithmus.

Version 2.0

Die Beschreibung von strings ist für alle Versionen (1.0B, 1.0C, 2.0) gültig.

Beispiel

In der Datei *programm.c* steht folgende C-Funktion:

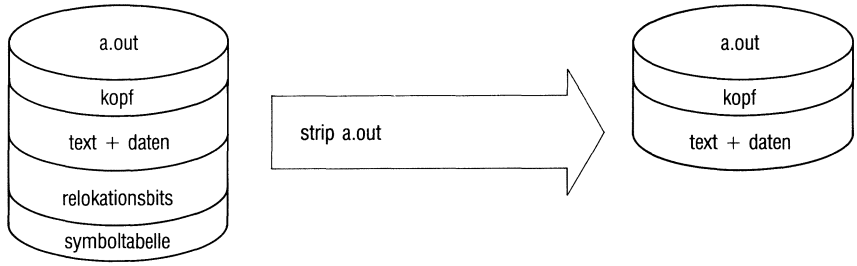
```
/* Beispielfunktion zum Suchen von Strings in Objektfiles */

char *kundenname(n)     /* Ergebnis: Name des n-ten Kunden */
int n;
{
    static char *kunde[]=
          {
                "Gudrun Falkner",
                "Petra Groeger",
                "Peter Hug",
                "Roland Meier",
                "Andreas Prott",
                "Andreas Schneider",
                "Andreas Stoll"
          };

    return(kunde[n]);
}

$
$
```

Auf der Standard-Ausgabe wird ausgegeben:

```
      58 Gudrun Falkner
      73 Petra Groeger
      87 Peter Hug
      97 Roland Meier
     110 Andreas Prott
     124 Andreas Schneider
     142 Andreas Stoll
$
```

> > > > xd

Symboltabellen und Relokationsbits entfernen - remove symbols and relocation bits



strip entfernt die Symboltabellen und Relokationsbits, die normalerweise in Objektmodulen und ausführbaren Programmen stehen.

Die Dateien, die strip als Eingabe erwartet, muß der Assembler oder Binder erstellt haben. strip verändert direkt die Dateien und gibt nichts auf der Standard-Ausgabe aus.

strip hilft Speicherplatz sparen, denn die Dateigröße verringert sich erheblich, wenn eine Datei mit strip behandelt wird.

strip sollte nur auf ein fehlerfreies Programm angewendet werden, da eine Fehlersuche ohne Symboltabelle schwieriger ist.

Die Wirkung von strip entspricht der des Schalters -s bei ld.

strip␣datei...

datei — *datei* sollte der Name eines Objektmoduls oder eines ablauffähigen Programms sein.

Version 2.0

Die Beschreibung von strip ist für alle Versionen (1.0B, 1.0C, 2.0) gültig.

Beispiel

In der Datei *programm.c* steht folgendes C-Programm:

```
main()
{
      printf("Hallo Peter, hier spricht Andy!\n");
}

$
$
-rwxrwxr-x 1 andreas    6316 Mar 26 13:17 a.out
$
$
-rwxrwxr-x 1 andreas    4648 Mar 26 13:18 a.out
$
```

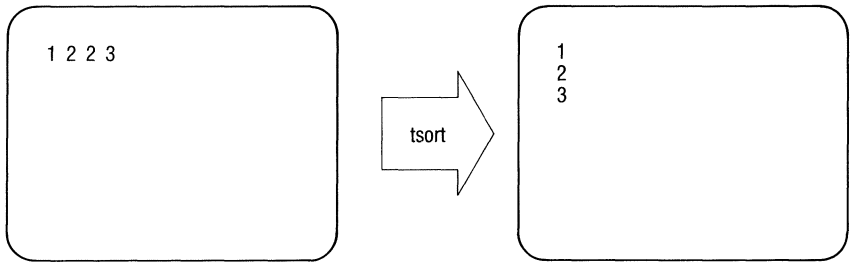
Mit Symboltabelle hat das übersetzte Programm eine Größe von 6316 Byte. Ohne Symboltabelle ergibt sich die Dateigröße 4648 Byte. Die Angaben beziehen sich auf die Version 1.0B. Bei Version 1.0C hat a.out die Standard-Schutzbiteinstellung `-rwx--x--x`.

Dateien

/tmp/stm? Zwischendateien

> > > > ld

Topologisch sortieren - topological sort



tsort versucht aus (partiell) geordneten Paaren von Wörtern eine (total) geordnete Liste der Wörter zu erstellen.

tsort erwartet als Eingabe eine gerade Anzahl von Wörtern. Die Wörter können beliebige druckbare ASCII-Zeichenreihen sein, die nicht leer sein und kein Leer- oder Tabulatorzeichen enthalten dürfen. Ein Wort muß durch ein oder mehrere Leerzeichen vom nächsten getrennt sein. Je zwei Wörter bilden ein Paar.

- Sind die beiden Wörter eines Paares verschieden, so bedeutet das, daß tsort das erste Wort auch in der sortierten Liste vor dem zweiten Wort einordnen soll.
- Wenn ein Paar aus zwei gleichen Wörtern besteht, so soll dieses Wort in die Liste eingefügt werden. An welcher Stelle es stehen soll, wird nicht festgelegt.

tsort schreibt die sortierte Liste auf die Standard-Ausgabe.

Sie können mit tsort Objektmodule innerhalb einer Bibliothek so ordnen,

- daß klar wird, welcher Modul von welchem abhängt d.h.
- daß ein Objektmodul, der Funktionsaufrufe und Zuweisungen an externe Variablen enthält, vor den Objektmodulen steht, die die entsprechenden Funktions- und Variablendefinitionen enthalten.

tsort[␣datei]

datei — In der Datei *datei* muß eine gerade Anzahl von Wörtern stehen, die jeweils durch Leerzeichen, Tabulatorzeichen oder "Neue Zeile" voneinander getrennt sind.
Wird keine Datei angegeben, so erwartet tsort, daß Sie die Wortpaare auf der Standard-Eingabe eingeben.

Hinweis

tsort arbeitet mit einem Algorithmus, bei dem Rechenzeit und Speicherplatz quadratisch anwachsen mit der Anzahl der Wörter. Sie sollten daher tsort nur zum Ordnen von Dateien in Bibliotheken verwenden.

Fehlerdiagnose

- Wenn Sie eine ungerade Anzahl von Zeichenreihen, also keine Paare, eingeben, so gibt tsort folgende Fehlermeldung aus:

  ```
  $ tsort: ungerade Anzahl von Daten
  ```

- Entdeckt tsort einen Zyklus (z.B. a b b a), dann lautet die Fehlermeldung:

  ```
  tsort: Zyklus
  tsort: a
  tsort: b
  a
  b
  $
  ```

Version 2.0

Die Beschreibung von tsort ist für alle Versionen (1.0B, 1.0C, 2.0) gültig.

Beispiele

1. Wörter auf der Standard-Eingabe eingeben:

   1 soll vor 2 und 2 vor 3 stehen.

   ```
   $ tsort ↵
   1 2 2 3 ↵
   CTRL D
   ```

   tsort erstellt folgende geordnete Liste:

   ```
   1
   2
   3
   $
   ```

2. Namen aus einer Datei eingeben:

   In der Datei *datei* steht:

   ```
   datei1 datei1 datei3 datei4 datei4 datei1 datei2 datei2
   ```

   tsort wird aufgerufen und erstellt folgende Liste:

   ```
   $ tsort datei
   datei3
   datei4
   datei1
   datei2
   $
   ```

3. Eine Bibliothek ordnen (siehe lorder, 2. Beispiel):

   ```
   $
   $
   $
   ```

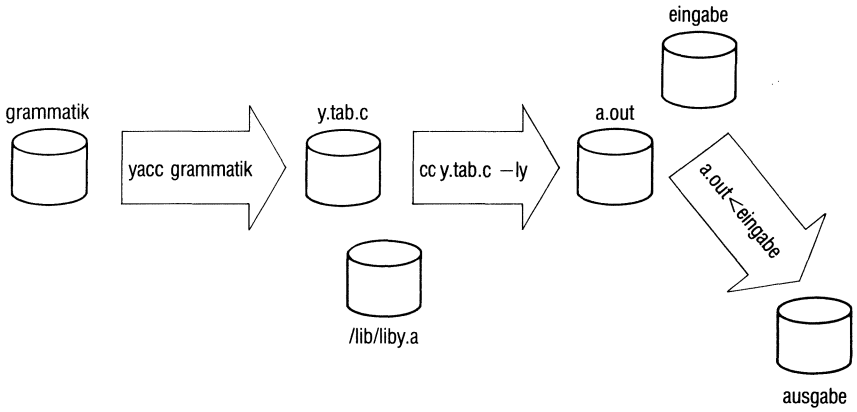
   lorder ordnet die Dateien, deren Namen mit ".o" enden (Objektmodule) paarweise vor. Die Ausgabe von lorder entspricht genau der Eingabe, die tsort erwartet. ar erstellt eine Bibliothek, in der die sortierten Objektmodule stehen und die ld nur einmal durchsuchen muß.

> > > > ar, lorder, ld, ranlib

Tastencode hexadezimal ausgeben

Die Beschreibung von xcho steht im Kapitel 3.

Parser generieren - yet another compiler-compiler



yacc generiert C-Programme, die einen Eingabetext analysieren und bearbeiten. Die Analyse betrifft die Zeichen, aus denen der Text besteht. Im Gegensatz zu lex, kann das yacc-Programm eine komplexe Struktur des Eingabetextes bestimmen, also eine größere Klasse von Eingabesprachen erkennen und weiterverarbeiten.

yacc kann u.a. den Parser für einen Übersetzer konstruieren. Zusammen mit einem lex-Programm für die lexikalische Analyse, werden damit dem Benutzer die wichtigsten Hilfsmittel bereitgestellt, die zum Übersetzerbau nötig sind.

Mit yacc wurden z.B. realisiert:

- Übersetzer für die Sprachen C, APL, Pascal, Ratfor, etc.
- verschiedene Tischrechner und Photosatzdrucker,
- ein Fortran-Debugger,
- ein "Document Retrieval System".

Um einen Übersetzer für eine bestimmte Eingabe(-sprache) zu konstruieren, müssen Sie u.a. Werkzeuge für folgende Aufgaben bereitstellen:

- eine Funktion, die die Eingabe, Zeichen für Zeichen, einliest und in bestimmte Einheiten (Token) zerlegt (**lexical Analyzer, Scanner**).
- eine Funktion, die die Struktur der Eingabeeinheiten erkennt und entsprechende Aktionen ausführt, wenn eine bestimmte Struktur auftritt (**Parser**). Dabei legt eine Grammatik die Struktur der Eingabe fest. Eine Eingabe soll nur dann vom Übersetzer akzeptiert werden, wenn sie den Grammatikregeln entspricht.

  Die Aktionen können u.a. die eigentliche Übersetzung ausführen, d.h. sie übersetzen die Eingabe in die Ausgabesprache.
- eine Funktion, die Fehler in der Eingabe behandelt.

yacc[␣-schalter...]␣datei

schalter

v — Zusätzlich zum C-Programm y.tab.c, gibt yacc in der Datei y.output eine Beschreibung der Übergangs-Tabellen des Parsers (Parsing-Tabellen) aus. Dabei berichtet yacc auch über alle Mehrdeutigkeiten, die in der Grammatik auftreten.

d — Es wird eine Datei y.tab.h erstellt, die #define-Anweisungen enthält. In den #define-Anweisungen werden die Token mit ihrer Tokennummer, die yacc oder der Benutzer definiert hat, in Verbindung gesetzt. Damit können auch andere Quell-Code-Dateien als y.tab.c auf die Tokennummern zugreifen.

Wenn für den Parser-Wertekeller andere Typen als integer vereinbart sind, dann steht auch die entsprechende typedef-Anweisung in y.tab.h.

datei — *datei* ist der Name der Datei, in der das yacc-Quell-Programm steht. Nach den Angaben im yacc-Quell-Programm schreibt yacc ein C-Programm in die Datei y.tab.c. Im C-Programm (yacc-Programm) wird u.a. eine Funktion yyparse definiert.

Wie arbeitet yacc?

yacc erwartet als Eingabe ein yacc-Quell-Programm, das in einem bestimmten Format geschrieben sein muß. Der Benutzer legt im yacc-Quell-Programm fest:

- die Struktur der Eingabe(-sprache), d.h. die **Grammatik**, die der Parser erkennen soll,
- Aktionen, die ausgeführt werden, wenn der Parser eine entsprechende Struktur findet,
- eine Funktion yylex, die die Eingabezeichen liest, nach Token vorgruppiert und für die Strukturbestimmung vorbereitet.
  yylex führt also die lexikalische Analyse aus (lex kann diese Funktion erstellen).

yacc erstellt aus diesen Angaben ein C-Programm, in dem ein Parser (yyparse) definiert wird. Genauer gesagt, erzeugt yacc die Übergangstabellen für einen Kellerautomaten, der nach einem LR(1)-Parsing-Algorithmus arbeitet. Eingabeeinheiten werden solange in dem Keller abgespeichert, bis sie eine Struktur bilden. Die Eingaben, die der Struktur angehören, werden aus dem Keller geholt und die Struktur wird abgespeichert.

Die Klasse der Grammatiken, die yacc bearbeiten kann, ist eine Teilmenge der Chomsky-2-Grammatiken (die LALR(1)-Grammatiken). Die Grammatiken dürfen bestimmte Mehrdeutigkeiten enthalten; mit Präzedenzregeln können Sie Mehrdeutigkeiten auflösen.

yacc schreibt das C-Programm, in dem die Funktion yyparse definiert wird, in die Datei y.tab.c. yyparse ist eine Integer-Funktion, die die Funktion yylex jedes Mal aufruft, wenn die nächste Eingabeeinheit eingelesen werden muß.

yyparse hat den return-Wert 0, wenn die Eingabe grammatikalisch richtig beendet und vom Parser akzeptiert wird. Falls ein Fehler in der Eingabe auftritt und nicht erfolgreich behoben werden kann, dann hat yyparse den return-Wert 1.

Sie müssen (neben yylex) noch folgende Funktionen im yacc-Quell-Programm definieren oder mit y.tab.o zusammenbinden:

- main, eine Hauptfunktion, die yyparse aufruft,
- yyerror, eine Funktion zur Fehlerbehandlung.

Einfache Versionen von main und yyerror stehen in der yacc-Bibliothek /lib/liby.a. Bei größeren Projekten ist es besser, eigene Versionen der Bibliotheksfunktionen zu verwenden.

Einfache Versionen von yyerror und main lauten z.B.:

```
main(){
        return(yyparse());
      }
```

und

```
#include <stdio.h>
yyerror(s) char *s; {
           fprintf(stderr, "%s\n", s);
      }
```

s ist die Adresse einer Zeichenfolge, die den Fehler beschreibt, z.B. "Syntaxfehler". Die Zeichenfolge muß mit dem Nullbyte abgeschlossen sein.

lex und yacc

In dem Programm, das lex erstellt, wird eine Funktion yylex definiert. Die Funktion, die die lexikalische Analyse bei yacc übernimmt, muß auch yylex heißen. lex und yacc passen also gut zusammen.

Normalerweise wird yylex von einer main-Funktion aufgerufen, die in der lex-Bibliothek steht. Wenn Sie yacc zusammen mit lex verwenden, dann erfolgt der Aufruf von yylex in dem Parser, den yacc bereitstellt.

Bei Zusammenarbeit von lex und yacc, sollte jede lex-Regel mit der Aktion

```
return(TOKEN);
```

enden und einen passenden Wert für *TOKEN* zurückgeben.

Das C-Programm, das lex erstellt, übersetzen Sie am besten als Teil des Programms, das yacc erstellt. Dies geschieht, indem Sie im letzten Abschnitt eines yacc-Quell-Programms (Programmteil) die Zeile

```
#include "lex.yy.c"
```

einfügen.

Steht das yacc-Quell-Programm z.B. in der Datei *grammatik* und das lex-Quell-Programm in der Datei *regeln*, dann kann ein Aufruf so aussehen:

```
$
$
$
$
```

Die yacc-Bibliothek muß vor der lex-Bibliothek gebunden werden, um ein Hauptprogramm zu haben, daß den Parser aufruft. Die Aufrufe von yacc und lex können Sie vertauschen.

Sie müssen das lex- und das yacc-Programm einzeln übersetzen und dann zusammenbinden, wenn die Programme zu groß sind, um zusammen übersetzt zu werden.

Wenn Sie lex und yacc zusammen verwenden, dann kann der Scanner das Erkennen von Chomsky-3-Strukturen (z.B. Identifikatoren) und der Parser das Erkennen von Chomsky-2-Strukturen (z.B. Anweisungen) und das Übersetzen übernehmen.

Wie arbeitet der Parser?

Wenn das yacc-Programm und die oben genannten anderen Funktionen richtig übersetzt und gebunden werden, dann steht in a.out ein ausführbares Programm. Das Programm behandelt einen Text, der auf der Standard-Eingabe bereitgestellt wird (den **Eingabestrom**), folgendermaßen:

- Das Hauptprogramm main ruft yyparse auf. yyparse ruft yylex auf.
- yylex liest soviele Zeichen des Eingabestroms ein, bis diese Zeichen eine Zeichenreihe bilden, die yylex erkennt. Wenn yylex mit lex erstellt wurde, werden also solange Zeichen eingelesen, bis sie eine Zeichenreihe darstellen, die zu einem regulären Ausdruck paßt.

  Die Zeichenreihen oder die Namen der Zeichenreihen, die yylex erkennt, sind die **Token**, **Terminalzeichen** oder **Terminalsymbole**. Ein Token kann der Name einer Zeichenreihe oder ein Einzelzeichen sein. Die Token stellen die Basiselemente der Eingabesprache dar. yylex meldet yyparse, welches Terminalsymbol gefunden wurde, bzw. ob ein Fehler aufgetreten ist.

- yyparse verarbeitet das Terminalsymbol, d.h. es sucht die Struktur, zu der das Terminalsymbol (eventuell zusammen mit vorher aufgetretenen Terminalsymbolen oder Strukturen) paßt und führt die Aktion aus, die bei der gefundenen Struktur angegeben ist.

  Die Namen der Strukturen sind die **Nicht-Terminalzeichen**, **Nicht-Terminalsymbole** oder **syntaktischen Variablen**.

  Die Aktionen bestimmen auch, ob das Programm a.out eine Ausgabe erstellt und auf welchem Ausgabemedium ausgegeben wird.

  yyparse veranlaßt yylex das nächste Terminalsymbol einzulesen.

- Wenn ein syntaktischer Fehler auftritt, wird er behandelt.

- Um das Ende eines Eingabestroms zu erkennen und dem Parser zu melden, muß yylex wissen, welches Terminalsymbol das "Ende der Eingabe" signalisiert. "Ende der Eingabe" kann z.B. "Dateiende" sein.

  Ein Eingabestrom wird akzeptiert, wenn er (ohne das Token "Ende der Eingabe" mitzubehandeln) die Struktur des "Startsymbols" hat.
  Der Benutzer muß genau ein **Startsymbol** definieren.
  Wird dem Parser "Ende der Eingabe" gemeldet, dann kehrt er zum Programm zurück, das ihn aufgerufen hat. In den meisten Fällen dürfte das die main-Funktion sein.

Beispiel für einen yacc-Aufruf

Zusammenfassend wird die Kommandofolge angegeben, mit der Sie einen Eingabestrom von einem yacc-Programm bearbeiten lassen können. In der Datei *grammatik* steht ein yacc-Quell-Programm. yylex wird im yacc-Quell-Programm definiert. Sie verwenden die einfachen Versionen von main und yyerror in der yacc-Bibliothek. Der Eingabestrom steht in der Datei *eingabe*. Der Aufruf lautet:

```
$ yacc grammatik
$ cc y.tab.c -ly
$ a.out < eingabe
```

Format eines yacc-Quell-Programms

Das allgemeine Format eines yacc-Quell-Programms sieht so aus:

```
[Deklaration...]
%%
Regel...
[%%]
[Programm...]
```

Deklarationen und Regeln müssen im entsprechenden Format geschrieben sein.

Ein Teil der Regeln kann auch die Fehlerbehandlung beschreiben, z.B. ob und wie ein Eingabestrom nach einem Fehler weitereingelesen und behandelt werden soll.

Im Programmteil können Sie C-Funktionen definieren,

- die im yacc-Programm aufgerufen werden (z.B. in Aktionen),
- die Teilfunktionen eines Übersetzers sind (z.B. yylex),
- die als Hauptfunktion fungieren sollen (z.B. main).

Allgemein gilt:

- Das Fluchtsymbol in einem yacc-Quell-Programm ist das Zeichen "%".
- yacc ignoriert Leer-, Tabulator- und "Neue Zeile"-Zeichen. Die Zeichen dürfen aber nicht den Teil eines Namens oder eines reservierten Symbols bilden, das aus mehreren Zeichen besteht.
- Kommentare können überall dort vorkommen, wo Namen stehen können (siehe unter "Deklarationen in yacc-Quell-Programmen" und "Regeln in yacc-Quell-Programmen").
  Die Kommentare müssen, wie in C, zwischen den Zeichen "/*" und "*/" stehen.
- Die Namen von Terminal- und Nicht-Terminalzeichen können beliebig lange Zeichenreihen sein. Sie können aus Buchstaben, Ziffern, und den Zeichen "_" und "." bestehen, dürfen aber nicht mit einer Ziffer beginnen.

– Namen, die yacc intern benutzt (z.B. "error" bei der Fehlerbehandlung) oder die in C reserviert sind (z.B. "if", "while"), können zu Schwierigkeiten führen.
Reservierte Namen als Token(namen) bereiten u.a. Probleme bei der lexikalischen Analyse und sollten vermieden werden.
yacc-interne Namen sollten Sie nur so verwenden, wie yacc es erwartet.

Deklarationen in yacc-Quell-Programmen

Im Deklarationsteil eines yacc-Quell-Programms können Sie folgende Anweisungen, Deklarationen und Definitionen angeben:

- %{
  text
  %}

  text — Der gesamte Text, der zwischen Zeilen steht, die nur "%{" und "%}" enthalten, wird (wie bei lex) in das C-Programm kopiert, das yacc erstellt. Dabei gilt der kopierte Text als global zu allen sonstigen Funktionen im C-Programm. Aktionen und Benutzerfunktionen können auf Funktionen und Variablen zugreifen, die hier definiert und deklariert werden.

  Wie bei lex, können Sie auf diese Weise Deklarationen, Definitionen, include-Anweisungen usw. an das C-Programm übergeben. Die Begrenzungszeilen werden nicht kopiert.

- %**token**␣NAME␣[nummer]...

  NAME — *NAME* ist ein Token(name), also ein Name für eine Zeichenreihe, die yylex erkennen und deren Tokennummer yylex an den Parser übergeben soll.

  Die Token oder Tokennamen müssen explizit als solche definiert werden. Üblicherweise bestehen die Namen von Token aus Großbuchstaben, die von Nicht-Terminalsymbolen aus Kleinbuchstaben. Nicht-Terminalsymbole werden nicht definiert, sie müssen aber mindestens einmal als Name einer Struktur auftreten (siehe unter "Regeln in yacc-Quell-Progrogrammen").

nummer — *nummer* muß eine ganze nicht-negative Zahl sein. *nummer* bezeichnet die Tokennummer, die *NAME* haben soll (siehe "Lexikalische Analyse", Die Tokennummer).

␣ — Als Trennzeichen "␣" können Sie beliebig viele Leer- oder Tabulatorzeichen angeben.

- %**start**␣symbol

  symbol — *symbol* ist der Name einer Struktur, also ein Nicht-Terminalsymbol.

  Genau ein Nicht-Terminalsymbol kann als Startsymbol deklariert werden. Das Startsymbol beschreibt die allgemeinste Struktur, die in der Grammatik vorkommt. Fehlt eine Start-Deklaration, dann wird die linke Seite der ersten Grammatikregel (das ist der Name der ersten Struktur), die im Regelteil steht, als Startsymbol genommen.

- %**left**␣TOKEN...
  %**right**␣TOKEN...
  %**nonassoc**␣TOKEN...

  TOKEN — *TOKEN* ist ein Tokenname oder ein Einzelzeichen. Ein Einzelzeichen muß zwischen Apostroph "'" stehen, also z.B. '+'.

  Token, die in derselben Zeile stehen, haben denselben Vorrang und dieselbe Assoziativität. Die Reihenfolge der Zeilen gibt den Vorrang der Token an. Die Zeilen müssen nach aufsteigendem Vorrang geordnet sein (siehe "Wie können Präzedenzen angegeben werden?").

- %**union** {
  unionkomponente...
  }

  unionkomponente

  unionkomponente ist eine Komponentenvereinbarung, wie sie normalerweise in einem C-Quell-Programm vorkommt, also z.B.

  ```
  double value;
  ```

  Mit dieser Vereinbarung können Sie die Typen der Elemente des Wertekellers (also von yylval und yyval) festlegen.
  Normalerweise sind die Werte vom Typ integer und müssen nicht explizit vereinbart werden.

  Siehe "yacc für Fortgeschrittene".

- %**type**␣ < value > ␣symbol...

  symbol — Die syntaktische Variable (linke Seite einer Regel) *symbol* soll vom selben Typ wie *value* sein.

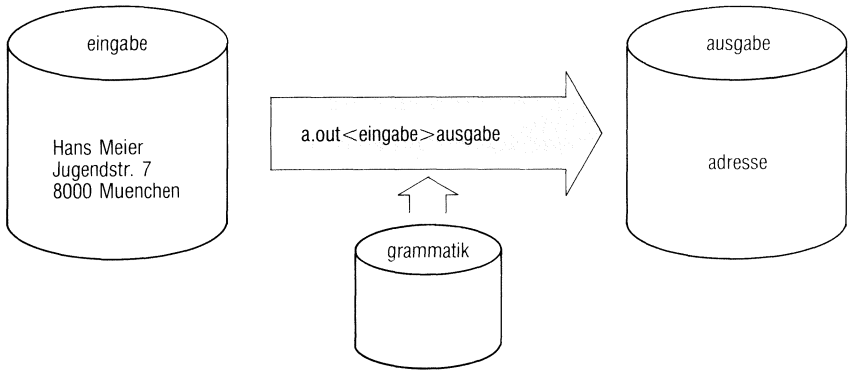
  value — *value* ist der Name einer Unionkomponente, die mit %union im Deklarationsteil vereinbart wurde.

Beispiel

```
%token VORNAME 301 FAMILIENNAME 302 STRASSE HAUSNUMMER POSTLEITZAHL ORT
```

Der Parser soll die Struktur einer Adresse in einem Eingabestrom erkennen. Der Scanner soll die Bestandteile der Adresse finden. Er liest jeweils soviele Zeichen des Eingabestroms bis die Zeichenreihe zu einem der angegebenen Token paßt und gibt die entsprechende Tokennummer zurück. *VORNAME* und *FAMILIENNAME* haben Tokennummer 301, bzw. 302. yacc bestimmt die Tokennummern von den restlichen Token.

Weitere Beispiele stehen in den Abschnitten "Regeln in yacc-Quell-Programmen", "yacc für Fortgeschrittene" und "Wie können Präzedenzen angegeben werden?"



Regeln in yacc-Quell-Programmen

Die **Regeln** sind Produktionsregeln der Grammatik, die die Eingabesprache beschreibt. In diesen Grammatikregeln werden Namen für Terminalzeichen (Token) und Namen für Nicht-Terminalzeichen (syntaktische Variablen) in Beziehung gesetzt. Jede Regel beschreibt eine erlaubte Struktur und gibt dieser Struktur einen Namen (in diesem Fall also den Namen einer syntaktischen Variablen). Die Token werden vom Scanner yylex im Eingabetext analysiert und dem Parser gemeldet. Dieser stellt anhand der Grammatikregeln fest, welche Struktur vorliegt.

Bei den Regeln können Sie Aktionen angeben, die auszuführen sind, wenn die entsprechende Struktur oder ein Teil der Struktur im Eingabestrom erkannt wird.

Die Regeln und die Aktionen schreiben Sie jeweils in verschiedene Zeilen, um das yacc-Quell-Programm leichter lesen und ändern zu können.

Eine Regel muß folgendes Format haben:

symbol : rumpf;

symbol — *symbol* ist der Name der Struktur, die nach dem Zeichen ":" in der Regel folgt. *symbol* ist also ein Name für ein Nicht-Terminalzeichen. *symbol* heißt auch "linke Seite der Regel".

rumpf — *rumpf* besteht aus der Struktur, die *symbol* heißt, und aus Aktionen, die auszuführen sind, wenn die Struktur oder ein Teil der Struktur im Eingabestrom vorkommt.

Eine *Struktur* ist eine Folge, die aus beliebig vielen

- Tokennamen,
- Namen für syntaktische Variable,
- Einzelzeichen des Zeichenvorrats

bestehen kann.

Die Tokennamen, die Namen für syntaktische Variable und die Einzelzeichen sind die *Komponenten der Struktur.* Zwischen den einzelnen Komponenten können beliebig viele Leerzeichen stehen. Die Leerzeichen werden von yacc ignoriert.

Die Aktionen werden unter "Aktionen in yacc-Quell-Programmen" genauer beschrieben.

Die *Einzelzeichen* müssen zwischen Anführungszeichen "'" stehen. Die Einzelzeichen sind Zeichen, die explizit im Eingabestrom vorkommen müssen und von yylex an den Parser übergeben werden. Sie sind also auch Token, haben aber keinen Namen und müssen nicht als Token im Deklarationsteil ausgewiesen werden.

Es liegt beim Benutzer, ob er bestimmte Eingabezeichenreihen als Token deklariert, und von yylex den Tokennamen (Tokennummer) melden läßt, wenn die entsprechende Zeichenreihe im Eingabestrom vorkommt. Oder ob er die Zeichenreihe als Folge von Einzelzeichen im Rumpf einer Regel beschreibt. yylex übergibt die Einzelzeichen (genauer die Tokennummer der Einzelzeichen) an den Parser.

Als Einzelzeichen können, wie in C mit "\" als Fluchtsymbol, auch folgende Zeichen angegeben werden:

```
'\n'      Neue Zeile
'\r'      Return
'\''      Apostroph
'\\'      Gegenschrägstrich
'\t'      Tabulator
'\b'      Rücktaste
'\f'      neue Seite
'\xxx'    Oxxx, d.h. die Oktalzahl xxx
```

Die ASCII-Null, '\0' oder 0, darf nie in Regeln verwendet werden, da die Tokennummer 0 für das Token "Ende der Eingabe" reserviert ist.

;

Jede Regel muß mit einem Strichpunkt abgeschlossen werden.

Haben mehrere Strukturen denselben Namen, d.h. mehrere Regeln dieselbe linke Seite, dann müssen Sie die linke Seite nicht jedes Mal wiederholen. Stattdessen können Sie das Wiederholungszeichen "|" verwenden. Solche Regeln haben das Format:

```
symbol : rumpf1
       | rumpf2
       ...
       | rumpfn
       ;
```

Der Strichpunkt ";" muß vor dem Wiederholungszeichen "|" weggelassen werden.

Um yacc-Quell-Programme leichter lesen und ändern zu können, sollten Sie

- alle Regeln mit derselben linken Seite nacheinander auflisten,
- die linke Seite nur einmal angeben und
- den Strichpunkt in eine extra Zeile am Ende schreiben.

Beispiel

Die Struktur des Datums beschreiben:

Die folgenden Grammatikregeln erkennen bestimmte Datumsangaben in der Eingabesprache, wie z.B. "5.7.1973" oder "7.Februar1930":

```
%token JAHR
%start datum
%%
datum   :   tag '.' zmonat '.' JAHR
        |   tag '.' wmonat JAHR
        ;

tag     :   '1'
        |   '2'
        ...
        |   '3' '1'
        ;

zmonat  :   '1'
        |   '2'
        ...
        |   '1' '2'
        ;

wmonat  :   'J' 'a' 'n' 'u' 'a' 'r'
        |   'F' 'e' 'b' 'r' 'u' 'a' 'r'
        ...
        |   'D' 'e' 'z' 'e' 'm' 'b' 'e' 'r'
        ;
```

Im Deklarationsteil wird *JAHR* als Terminalsymbol deklariert, *datum* als Startsymbol. yylex, der Scanner, übergibt an den Parser die Meldung, welches Terminalsymbol er findet. *tag* ist ein Nicht-Terminalsymbol und yylex übergibt die einzelnen Ziffern als Einzelzeichen an den Parser. Sie können die Regeln noch so erweitern, daß auch geprüft wird, ob ein bestimmter Tag überhaupt existiert, wie z.B. der 30. Februar 1978.

Es liegt bei Ihnen, ob Sie z.B. auch *tag*, *zmonat* und *wmonat* als Token deklarieren möchten. Der Scanner übernimmt dann Aufgaben des Parsers.

Aktionen in yacc-Quell-Programmen

Bei jeder Regel kann der Benutzer Aktionen angeben, die auszuführen sind, wenn die Struktur (oder ein Teil der Struktur) der entsprechenden Regel im Eingabestrom vorkommt.

Die **Aktionen** sind C-Programmteile, die u.a. folgendes können:

- Werte zurückgeben,
- von yylex Werte für Terminalsymbole entgegennehmen,
- untereinander kommunizieren,
- Ein- und Ausgabe ausführen,
- andere Funktionen aufrufen,
- externe Vektoren und Variablen ändern.

Eine Aktion muß in geschweifte Klammern geschrieben werden, also

```
{ C-Programmteil }
```

Definitionen und Deklarationen, die global für alle Aktionen gelten sollen, können Sie im Deklarationsteil angeben. Die Definitionen müssen zwischen den Zeilen mit den Zeichen "%{" und "%}" stehen (siehe unter "Deklarationen in yacc-Quell-Programmen").

Aktionen müssen nicht am Ende einer Regel stehen. Sie können auch zwischen Struktur-Komponenten eingefügt und ausgeführt werden, bevor die Struktur ganz vom Parser bearbeitet wird.

Da der Parser, den yacc generiert, nur Variable benutzt, die mit "yy" beginnen, sollten Sie solche Variablen- und Funktionsnamen vermeiden oder so verwenden, wie yacc es erwartet.

Werte

Um die Kommunikation zwischen den Aktionen, yylex und dem Parser zu erleichtern, gibt es Pseudovariable, die mit dem Zeichen "$" beginnen und bestimmte **Werte** an den Parser zurückgeben.

In einer Regel haben sowohl die linke Seite als auch die Strukturkomponenten und Aktionen auf der rechten Seite einen Wert. Normalerweise ist der Wert eine ganze Zahl. Für diese Werte gilt:

- Der Wert einer Aktion im yacc-Quell-Programm ist
  - der Wert, der innerhalb derselben Aktion an die Pseudovariable $$ übergeben wird,
  - 0, wenn in der Aktion der Variablen $$ kein Wert zugewiesen wird.
- Der Wert eines Token (nicht zu verwechseln mit der Tokennummer) ist
  - der Wert, den die externe Variable yylval hat, wenn yylex die Tokennummer an den Parser übergibt.

    yylval hat den Anfangswert 0 und behält einen festen Wert, bis ein anderer Wert an yylval zugewiesen wird.
    Wenn yylex ein Token im Eingabestrom erkennt und (z.B. in einer Aktion im lex-Quell-Programm) an yylval einen Wert zuweist, dann hat das Token diesen Wert. Siehe auch "Lexikalische Analyse".
- Der Wert einer syntaktischen Variablen (der linken Seite einer Regel) ist
  - der Wert von $$, wenn auf der rechten Seite als letztes eine Aktion steht, in der an $$ ein Wert zugewiesen wird,
  - der Wert von $1, sonst.

Die Bedeutung der Pseudovariablen

$$ Eine Aktion kann an die Pseudovariable $$ einen Wert zuweisen.

$$ hat dann innerhalb der Aktion diesen Wert.
Wenn innerhalb einer Aktion kein Wert an $$ zugewiesen wird, dann hat $$ innerhalb der Aktion

- den Wert von $1, wenn die Aktion als letztes auf der rechten Seite steht,
- den Wert 0, sonst.

$1, $2, $3,...

Innerhalb einer Regel kann auf die Werte von früheren Aktionen und Strukturkomponenten derselben Regel zugegriffen werden. $1 hat den Wert der Komponente oder der Aktion, die als erste rechts vom Doppelpunkt steht, $2 den Wert der nächsten Komponente oder Aktion usw.

Beispiele

1. Beispiel für Werte:

   Bei der Regel

```
a : B C D
            {$$ = $2;}
            ;
```

   haben die Pseudovariablen folgende Werte:

   $$ hat den Wert von $2 (den Wert von *C*),
   $1 hat den Wert von *B*,
   $2 hat den Wert von *C*,
   $3 hat den Wert von *D*,
   $4 hat den Wert der Aktion, also von $$.

   Die Token *B*, *C*, und *D* haben den Wert, den die Variable yylval hat, wenn yylex die jeweilige Tokennummer an den Parser übergibt.

   Die syntaktische Variable *a* hat den Wert von $$.

   Würde die Regel

```
a : B C D
```

   lauten, dann hätte *a* den Wert von $1, also von *B*.

2. Beispiel einer Aktion:

```
a : '(' B ')'
              { printf("Hallo!\n");   }
```

Auf der Standard-Ausgabe wird

```
Hallo!
```

ausgegeben, sobald der Parser die Struktur '(' *B* ')' erkennt.

```
$1 hat den Wert von '(',
$2 hat den Wert von B,
$3 hat den Wert von ')',
$4 hat den Wert der Aktion, also 0, da keine Zuweisung an $$ erfolgt,
a hat den Wert von $1.
```

3. Beispiel für mehrere Aktionen innerhalb einer Struktur:

In der Datei *yacc.b* steht:

```
%token B C
%%
b : a C
        { printf("%d %d\n", $1, $2);}

a : '('
        { $$ = 5;}
     B
        { printf("%d\n", $$);}
    ')'
        { printf("%d %d %d %d %d %d\n", $1, $2, $3, $4, $5, $$);}
     B
        { printf("%d %d %d\n", $6, $7, $$);}
%%
#include "lex.yy.c"
```

In Datei *lex.b* steht:

```
        extern int yylval;
%%
B       return(B);
"("     {yylval = 9; return('(');}
")"     {yylval = 7; return(')');}
C       return(C);
.       ;
\n      ;
```

Nach den Kommandos

```
$ lex lex.b
$ yacc yacc.b
$ cc y.tab.c -ly -lln
$ a.out
```

und der Eingabe

```
(B)BC
```

folgt die Ausgabe

```
0
9 5 9 0 7 0
0 7 9
9 7
```

In der ersten Zeile steht der Wert, den $$ bei der zweiten Aktion von *a* annimmt. In der zweiten Zeile stehen die Werte von $1 bis $5 und der Wert von $$, den $$ in der dritten Aktion von *a* hat. Die nächste Zeile zeigt die Werte von $6, $7 und von $$ bei der letzten Aktion. In der letzten Zeile stehen die Werte der Variablen $1 und $2 innerhalb der Struktur *b*.

Lexikalische Analyse

Der Benutzer muß eine Funktion yylex zur Verfügung stellen, die die lexikalische Analyse des Eingabestroms übernimmt, d.h. eine Funktion, die

- den Eingabestrom Zeichen für Zeichen einliest,
- Token erkennt,
- Tokennummern und eventuell Werte an den Parser übergibt.

Diese Funktion yylex muß einen Integer-Wert, die **Tokennummer**, mit einer return-Anweisung zurückgeben. Die Tokennummer repräsentiert ein Token eindeutig. yylex kann im Programmteil definiert oder mit einer include-Anweisung eingefügt werden (siehe "lex und yacc"). Mit lex lassen sich Programme erstellen, die die lexikalische Analyse von vielen, aber nicht allen (z.B. Fortran), Eingabesprachen übernehmen können.

yylval ist eine Variable, mit der yylex und der Parser kommunizieren können:

yylval — Soll yylex außer der Tokennummer noch einen Wert zurückgeben, so müssen Sie diesen Wert der externen Variablen yylval zuweisen. Die Zuweisung muß von yylex ausgeführt werden. Sie geben z.B. in einer Aktion im lex-Quell-Programm an, welcher Wert mit yylval jeweils zurückgegeben werden soll.
yylval ist mit 0 vorbelegt und behält einen festen Wert, bis ein anderer zugewiesen wird. Der Wert eines Token ist der Wert, den yylval hat, wenn yylex an den Parser die Tokennummer übergibt (siehe "Aktionen in yacc-Quell-Programmen).

Die Tokennummer

Die Tokennummer, die einem Token zugewiesen wird, muß beim Parser und bei yylex dieselbe sein. Sie kann sowohl vom Benutzer als auch von yacc festgelegt werden. In beiden Fällen steht sie in der Datei y.tab.h, wenn yacc mit dem Schalter d aufgerufen wird.

yylex kann den Namen des Token zurückgeben ohne die Tokennummer zu wissen, denn die Tokennummer wird (von yacc) mit dem Tokennamen verknüpft wie bei einer #define-Anweisung (siehe y.tab.h).

Sie können im Deklarationsteil des yacc-Quell-Programms einem Token (das kann auch ein Einzelzeichen sein) eine Tokennummer zuweisen. Es genügt, wenn Sie beim ersten Vorkommen des Namens eines Token oder eines Einzelzeichens, direkt danach ein Leerzeichen und eine nicht-negative ganze Zahl schreiben, die die Tokennummer sein soll.

Wenn Sie keine Tokennummer zur Verfügung stellen, dann nimmt yacc als Tokennummer

- bei Einzelzeichen — den numerischen Wert des Zeichens im gegebenen Zeichenvorrat, z.B. im ASCII-Zeichenvorrat,
- bei Tokennamen — eine Zahl, die größer als 256 ist.

Tokennummern von verschiedenen Token müssen verschieden sein. Darauf ist besonders zu achten, wenn Sie nur einen Teil der Tokennummern selbst zuweisen und yacc die restlichen Tokennummern festlegt.

Die Tokennummer des Zeichens "Ende der Eingabe" muß 0 oder negativ sein. Der Benutzer kann sie nicht definieren.

Beispiel

Sie definieren im Deklarationsteil eines yacc-Quell-Programms das Terminalsymbol *ZIFFER* mit

```
%token ZIFFER 48
```

Die Tokennummer von *ZIFFER* soll 48 sein. yylex kann mit dem Parser kommunizieren ohne die Tokennummer zu wissen. yylex können Sie im Programmteil des yacc-Quell-Programms z.B. so definieren:

```
yylex() {
        extern int yylval;
        int c;
        ...
        c = getchar();                  /* Das nächste Zeichen des
                                           Eingabestroms einlesen */
        ...
        switch (c) {
        ...
        case '0' :
        case '1' :
        ...
        case '9' :
                    yylval = c-'0';     /* Wenn das eingelesene Zeichen
                                           eine Ziffer ist, dann soll
                                           zusätzlich zur Tokennummer noch
                                           der numerische Wert der Ziffer
                                           zurückgegeben werden */
                    return (ZIFFER);    /* yylex gibt die Tokennummer für
                                           ZIFFER zurück, wenn eine Ziffer
                                           eingelesen wird */
        ...
                   }
        ...
      }
```

Wie arbeitet der Parser intern?

Aus dem yacc-Quell-Programm erstellt yacc einen Parser, und zwar einen Kellerautomaten mit endlich vielen Zuständen. Das oberste Kellerelement, das Topelement, gibt den aktuellen Zustand an, in dem sich der Automat gerade befindet.

Die Aktionen des Kellerautomaten sollen als **Aktionen(P)**, Parser-Aktionen, bezeichnet werden um sie von den Aktionen zu unterscheiden, die der Benutzer im yacc-Quell-Programm angeben kann.

In Übergangs-Tabellen steht, welche Aktion(P) der Automat jeweils ausführt und in welchen Zustand der Automat als nächstes übergeht.

Die Übergangs-Tabellen stehen in der Datei y.output, wenn Sie yacc mit dem Schalter v aufrufen.

Der Folgezustand des Automaten ist abhängig

- vom aktuellen Zustand und
- eventuell von einem Token oder von einer syntaktischen Variablen.

  Das Token ist das nächste, das von yylex eingelesen und an den Automaten übergeben wird. Dieses Token heißt **lookahead token** oder **Vorausschau**.

  Eine syntaktische Variable beeinflußt dann den nächsten Zustand, wenn die gesamte rechte Seite einer Regel abgearbeitet ist, und durch die linke Seite ersetzt wird (siehe "reduziere"). Die syntaktische Variable auf der linken Seite zusammen mit dem aktuellen Zustand bestimmt den nächsten Zustand.

Die Zustände werden mit ganzen Zahlen bezeichnet. Der Startzustand ist 0. Am Anfang befindet sich im Keller nur dieser Zustand, noch kein Eingabezeichen ist eingelesen.

Zusätzlich zum Kellerspeicher für Zustände, arbeitet der Automat noch mit einem Kellerspeicher für Werte. In ihm werden die Werte abgespeichert, die die Strukturkomponenten und Aktionen haben, und durch den Wert der syntaktischen Variablen auf der linken Seite ersetzt, wenn die rechte Seite einer Regel abgearbeitet ist. Die Pseudovariablen $1, $2, $3... beziehen sich auf die Elemente des Wertekellers.

Der Zustands- und der Wertekeller sind Felder der Länge YYMAXDEPTH. Die Konstante YYMAXDEPTH ist mit dem Wert 150 vordefiniert. Sie können sie umdefinieren.

Wenn die rechte Seite einer Regel z.B. vier Strukturkomponenten und Aktionen hat und die Struktur der rechten Seite abgearbeitet ist, dann bezeichnet $4 das oberste, $3 das zweitoberste, $2 das drittoberste und $1 das viertoberste Element des Wertekellers. Wenn die gesamte rechte Seite abgearbeitet ist, werden die vier obersten Werte aus dem Wertekeller entfernt und der Wert der syntaktischen Variablen (auf der linken Seite) als oberstes Element in den Wertekeller gebracht.

Der Automat kennt fünf Aktionen(P):

- lies,
- reduziere,
- goto,
- akzeptiere,
- Fehler.

Aktionen(P)

lies

Bei "lies" sind immer eine Vorausschau und eine Zustandsnummer angegeben. Wenn die betreffende Vorausschau (genauer die Tokennummer der Vorausschau) von yylex an den Parser übergeben wird, dann soll der Zustand, dessen Nummer angegeben ist, der aktuelle Zustand werden. Der Zustandskeller wird also um ein Element verlängert. Das Token, das die Vorausschau bildete, wird jetzt vom Parser weiterverarbeitet. Es kann kein zweites Mal als Vorausschau dienen und von yylex bearbeitet werden.

Im Wertekeller wird der Wert der globalen Variablen yylval abgespeichert.

Beispiel

Der Automat ist im Zustand 5. Beim Zustand 5 ist die Aktion

KLINGEL lies 7

gegeben. Wenn die Vorausschau, die yylex an den Parser übergibt, *KLINGEL* ist, dann soll Zustand 7 zum aktuellen Zustand werden. Zustand 5 bleibt, wie alle vorher schon vorhandenen Zustände im Keller. Das Token *KLINGEL* ist damit abgearbeitet und kann nicht mehr als Vorausschau verwendet werden.

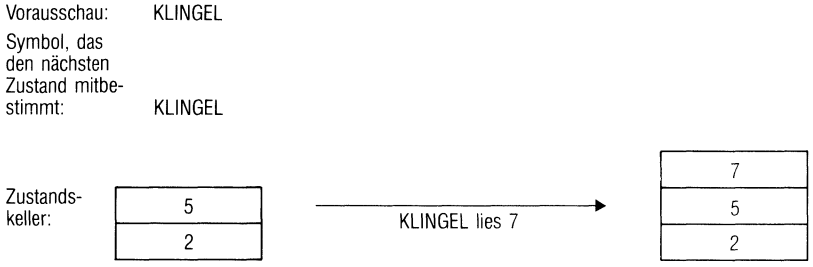


Bild 2-1 Aktion(P) "lies"

reduziere

Eine reduziere-Aktion(P) verhindert, daß der Keller überläuft. Wenn der Parser die gesamte rechte Seite einer Regel, d.h. die dort beschriebene Struktur, gesehen hat, dann ersetzt er die rechte Seite durch den Namen der syntaktischen Variablen, der auf der linken Seite der Regel steht.

Dabei werden aus dem Zustandskeller soviele Zustände entfernt, wie die Struktur auf der rechten Seite Komponenten hat. Der Zustand, der jetzt an oberster Stelle im Keller steht, und die linke Seite der Regel bestimmen, welche Folgeaktion(P) der Automat ausführen soll und in welchen Zustand er übergehen soll. Die Folgeaktion(P) ist eine goto-Aktion.
Vorausschau ist (meistens) weder bei "reduziere" noch bei "goto" nötig.

Aus dem Wertekeller werden genauso viele Einträge geholt, wie die rechte Seite der Regel Strukturkomponenten und Aktionen hat. Bei einer Folgeaktion(P) "goto" wird die globale Variable yyval im Wertekeller abgespeichert. yyval hat den Wert der syntaktischen Variablen, deren Struktur gefunden wurde ($1 bzw. $$).

Bei einer reduziere-Aktion(P) ist immer eine Zahl angegeben. Diese Zahl bezeichnet die Regel, nach der die Keller reduziert werden sollen. Die Regel steht in y.output direkt vor der reduziere-Aktion(P). Die Nummer der Regel ist in runde Klammern "(" und ")" eingeschlossen und steht unmittelbar nach der jeweiligen Regel.

Aktionen, die der Benutzer bei der Struktur im yacc-Quell-Programm angibt, werden ausgeführt, bevor Zustands- und Wertekeller reduziert werden.

Beispiel

Der Automat ist im Zustand 5. Im Zustand 5 seien die folgende Regel und die folgende Aktion(P) gegeben:

```
the : DER DIE DAS_ (9)
.     reduziere 9
```

Es soll nach der Regel 9 reduziert werden. Die Regel 9 hat auf der rechten Seite drei Komponenten (*DER*, *DIE*, *DAS*). Es werden daher drei Kellerelemente entfernt.

" _ " bedeutet, daß yylex die Strukturkomponenten links von " _ " bereits eingelesen und an den Parser weitergegeben hat.

Der Punkt "." bedeutet, daß diese Aktion(P) nur dann ausgeführt werden soll, wenn der Automat im aktuellen Zustand keine andere Aktion(P) anwenden kann.

Vom Zustands- und Wertespeicher werden die drei obersten Elemente entfernt.

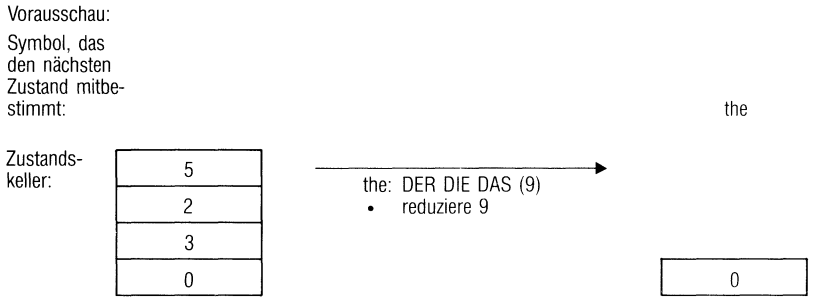


Bild 2-2 Aktion(P) "reduziere"

Auf eine reduziere-Aktion(P) folgt eine goto-Aktion(P). Es könnte also im Zustand 0 folgende Aktion(P) gegeben sein:

```
the     goto  15
```

goto

Bei einer goto-Aktion(P) sind eine Zustandsnummer und der Name einer syntaktischen Variablen angegeben. Wenn die Struktur abgearbeitet und durch ihren Namen ersetzt wurde (siehe reduziere), dann soll der Zustand mit der Nummer, die bei "goto" genannt wird, im Keller abgespeichert werden. Dieser Zustand wird also zum neuen Topzustand. Statt einer Vorausschau bestimmt die syntaktische Variable den Folgezustand.

Bei einer goto-Aktion(P) wird der Wert der globalen Variablen yyval im Wertekeller abgespeichert. yyval hat den Wert der syntaktischen Variablen, die bei der goto-Aktion(P) steht.

Beispiel

Der Automat sei im Zustand 0. Im Zustand 0 sei die folgende Aktion(P) gegeben:

```
the     goto  15
```

Der Automat geht in den Zustand 15 über. "the" bestimmt den nächsten Zustand. Der Automat verhält sich folgendermaßen:

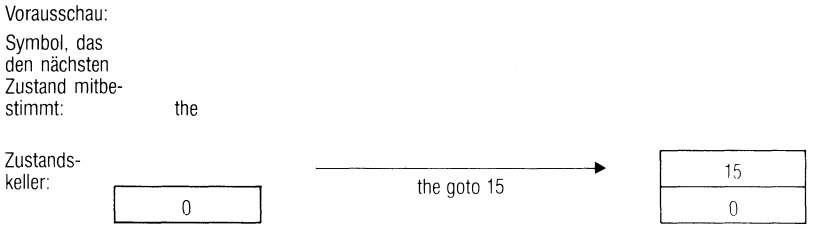


Bild 2-3 Aktion(P) "goto"

akzeptiere

Die akzeptiere-Aktion(P) zeigt an, daß der gesamte Eingabestrom eingelesen wurde und zu den Grammatikregeln paßt.
Die Vorausschau ist "Ende der Eingabe".
Der Parser akzeptiert die Eingabe und übergibt den Wert 0 an die Funktion, von der yyparse aufgerufen wurde. Der Wert 0 bestätigt das erfolgreiche Bearbeiten der Eingabe.

Beispiel

Der Automat ist im Zustand 1. Die Vorausschau ist das Zeichen "Ende der Eingabe", das mit $end bezeichnet wird. Im Zustand 1 ist die folgende Regel gegeben:

```
$end akzeptiere
```

Der Automat verhält sich wie folgt:

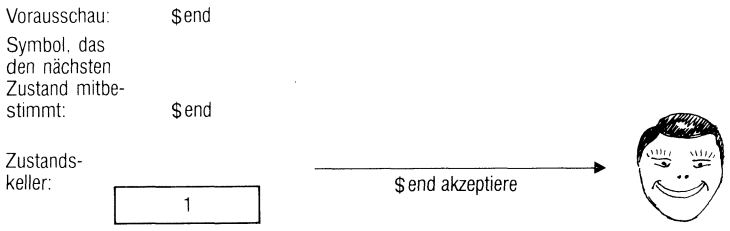


Bild 2-4 Aktion(P) "akzeptiere"

Fehler

Der Parser führt diese Aktion(P) aus, wenn im aktuellen Zustand aus der Vorausschau klar wird, daß im Eingabestrom ein Fehler vorhanden ist. Der Parser kann keine andere Aktion(P) ausführen, die den Angaben im yacc-Quell-Programm entsprechen würde. Der Parser führt die Aktion(P) "Fehler" aus, die Fehlerbehandlung setzt ein (siehe "Fehlerbehandlung").

Beispiel

Der Automat ist im Zustand 1. Im Zustand 1 sind nur die folgenden Regeln gegeben:

```
KLINGEL lies 5
. Fehler
```

Die Vorausschau ist *BIMMEL*. Es muß also ein Fehler in der Eingabe vorliegen. Der Automat arbeitet wie folgt:

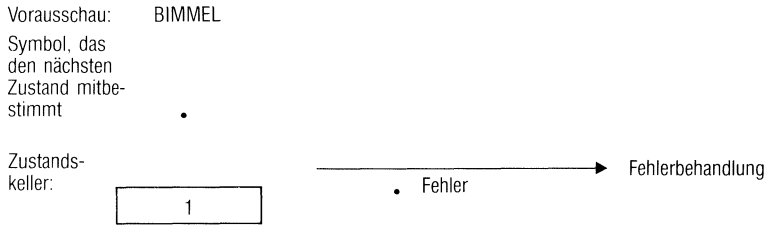


Bild 2-5 Aktion(P) "Fehler"

Beispiel

Der Teil eines yacc-Quell-Codes (ohne yylex) lautet:

```
%token ALLE MEINE ENTCHEN
%%
lied    : anfang ende
        ;
anfang  : ALLE MEINE
        ;
ende    : ENTCHEN
        ;
```

yacc schreibt die Übergangstabellen in lesbarer Form in die Datei y.output, wenn Sie yacc mit dem Schalter v aufrufen.

In der Datei y.output stehen folgende Übergangstabellen:

```
Zustand 0
        $accept : _lied $end

        ALLE  lies 3
        .  Fehler

        lied  goto 1
        anfang  goto 2

Zustand 1
        $accept :  lied_$end

        $end  akzeptiere
        .  Fehler

Zustand 2
        lied :  anfang_ende

        ENTCHEN  lies 5
        .  Fehler

        ende  goto 4

Zustand 3
        anfang :  ALLE_MEINE

        MEINE  lies 6
        .  Fehler

Zustand 4
        lied :  anfang ende_    (1)
```

```
        .  reduziere 1

Zustand 5
        ende : ENTCHEN_     (3)

        .  reduziere 3

Zustand 6
        anfang :  ALLE MEINE_     (2)

        .  reduziere 2
```

Das Zeichen " _ " zeigt an, was bereits gelesen oder behandelt wurde (links von " _ ") und was noch gelesen werden muß (rechts von " _ ").

Bei der Eingabe "Alle meine Entchen" erkennt yylex die Token *ALLE*, *MEINE* und *ENTCHEN*.

Der Automat arbeitet folgendermaßen:

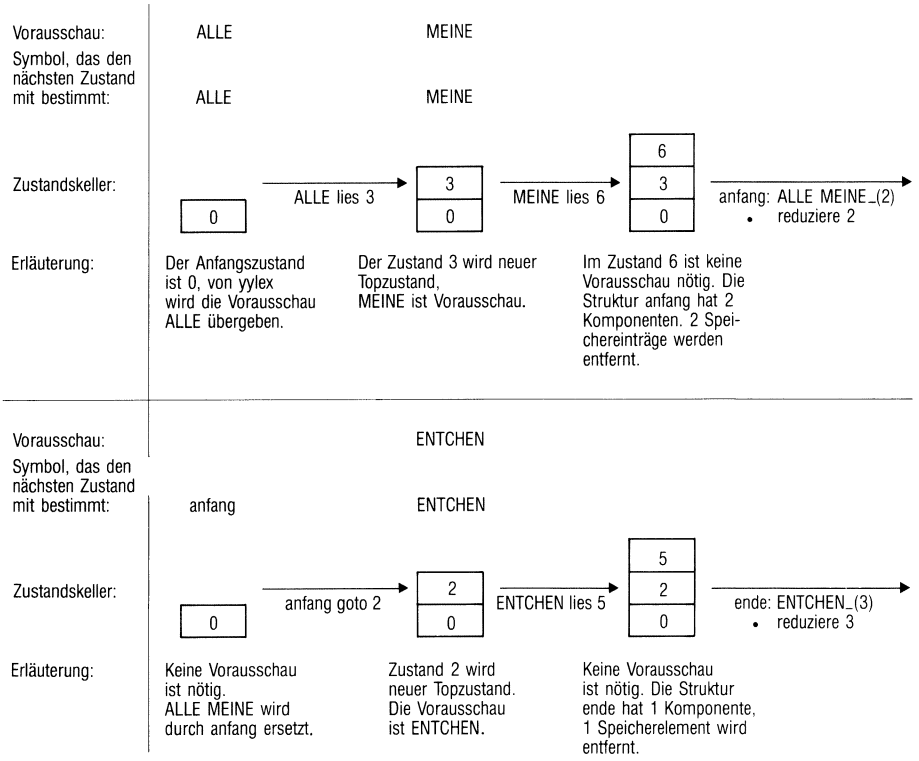


Bild 2-6 Aktionen(P) nach der Eingabe "Alle meine Entchen"

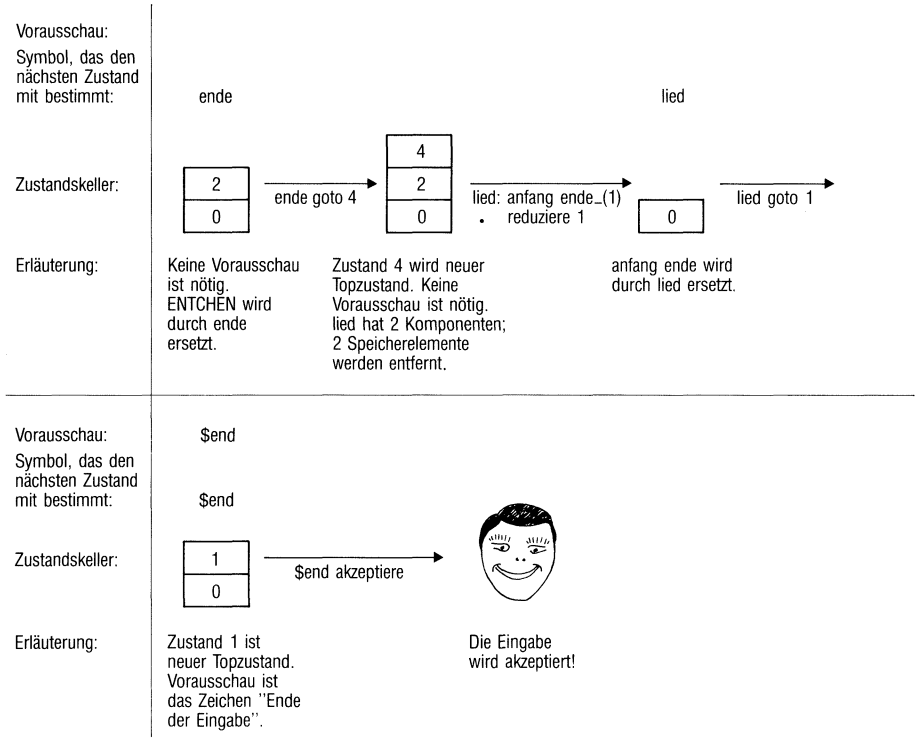


Bild 2-7 Aktionen(P) nach der Eingabe "Alle meine Entchen"

Wie werden mehrdeutige Grammatiken und Konflikte behandelt?

Eine Menge von Grammatikregeln heißt **mehrdeutig**, wenn es eine Eingabe gibt, die auf mehr als eine Weise strukturiert werden kann.

Beispiel

Eine Regel lautet:

```
ausdruck : ausdruck '-' ausdruck
```

Die Eingabe

```
ausdruck - ausdruck - ausdruck
```

kann auf zwei Arten strukturiert werden, nämlich:

```
(ausdruck - ausdruck) - ausdruck
```

oder

```
ausdruck - (ausdruck - ausdruck)
```

Im ersten Fall spricht man von **Links-Assoziation**; der Parser reduziert "ausdruck - ausdruck" und liest danach erst "- ausdruck" ein.

Im zweiten Fall spricht man von **Rechts-Assoziation**. Der Parser liest die gesamte Eingabezeile ein, speichert sie im Keller ab und bearbeitet den Keller nach der Last-In-First-Out-Methode. Statt der reduziere-Aktion(P) führt der Parser zuerst eine lies-Aktion(P) aus.

Die oben beschriebene Situation beschreibt einen **lies/reduziere-Konflikt**.

Sind bei derselben Eingabe mehrere gültige Reduktionen möglich, dann liegt ein **reduziere/reduziere-Konflikt** vor.

Bei Konflikten kann der Benutzer Regeln bereitstellen, die die Mehrdeutigkeiten beseitigen. Existieren keine solchen Regeln, dann verfährt der Parser, den yacc erstellt, wie folgt:

- Bei einem lies/reduziere-Konflikt wählt der Parser die lies-Aktion(P) aus.
- Bei einem reduziere/reduziere-Konflikt wählt der Parser die Reduktion, die im yacc-Quell-Programm als erste angegeben ist.

yacc meldet die Anzahl der lies/reduziere-Konflikte und der reduziere/reduziere-Konflikte und auf welche Art der Parser sie löst.

Wenn yacc mit dem Schalter v aufgerufen wird, dann werden noch die Nummern der Zustände und die Symbole angegeben, die den Konflikt verursachen.

Wie können Präzedenzen angegeben werden?

Die oben genannten Regeln zur Konfliktlösung sind nicht ausreichend um arithmetische Ausdrücke mit dem Parser zu behandeln.

Regeln, die den Vorrang und die Assoziativität von Token (z.B. Operatoren) angeben, ermöglichen auch die Verwendung von arithmetischen Ausdrücken in einer Eingabesprache.

Die Angaben zur Assoziativität und zum Vorrang von Token müssen im Deklarationsteil des yacc-Quell-Programms stehen. Sie müssen in einem der folgenden Formate geschrieben sein:

%**left**␣TOKEN...

%**right**␣TOKEN...

%**nonassoc**␣TOKEN...

TOKEN ist ein Tokenname oder ein Einzelzeichen, der bzw. das z.B. einen Operator bezeichnet.

Token, die in derselben Zeile stehen, haben denselben Vorrang und dieselbe Assoziativität. Die Reihenfolge der Zeilen gibt den Vorrang der Token an. Die Zeilen müssen nach aufsteigendem Vorrang geordnet sein.

Arithmetische Ausdrücke, in denen nacheinander mehrere Token vorkommen, die in derselben Zeile deklariert werden, sollen

- von links zusammengefaßt und behandelt werden, wenn die Deklarationszeile mit "%left" beginnt,
- von rechts zusammengefaßt und behandelt werden, wenn die Deklarationszeile mit "%right" beginnt,
- nicht korrekt sein, wenn die Deklarationszeile mit "%nonassoc" beginnt.

Tokennamen und Einzelzeichen, deren Assoziativität oder Vorrang deklariert wird, müssen Sie nicht mit

```
%token
```

deklarieren. Sie können eine solche Deklaration aber zusätzlich angeben.

Wenn Sie Vorrangregeln angeben wollen und ein einstelliger und ein zweistelliger Operator durch dasselbe Zeichen repräsentiert werden (z.B. "-"), aber unterschiedlichen Vorrang haben, dann muß das auf folgende Weise geschehen:

Im Deklarationsteil des yacc-Quell-Programms deklarieren Sie den Vorrang des zweistelligen Operators. Wie der einstellige Operator verwendet werden soll, geben Sie in einer Grammatikregel an. In dieser Regel muß außerdem noch

%**prec**␣TOKEN

stehen. Dabei ist *TOKEN* ein Tokenname oder ein Einzelzeichen, für den bzw. das bereits ein Vorrang deklariert ist.

Die Grammatikregel mit "%prec" soll denselben Vorrang haben wie der Operator, der mit *TOKEN* bezeichnet wird. "%prec TOKEN" muß in der Regel unmittelbar nach den Strukturkomponenten und vor einer Aktion oder einem Strichpunkt stehen.

Beispiel

Die folgenden Grammatikregeln beschreiben arithmetische Ausdrücke, in denen

- die zweistelligen Operatoren "*" und "/" Vorrang vor den zweistelligen Operatoren "+" und "-" haben,
- die vier Operatoren "+", "-", "*" und "/" linksassoziativ sind,
- die Operatoren "+", "-", "*" und "/" Vorrang vor dem Operator "=" haben,
- der Operator "=" rechtsassoziativ ist,
- der einstellige Operator "-" denselben Vorrang wie "*" haben soll.

```
%token NAME
%right  '='
%left   '+' '-'
%left   '*' '/'

%%
ausdruck  :  ausdruck '=' ausdruck
          |  ausdruck '+' ausdruck
          |  ausdruck '-' ausdruck
          |  ausdruck '*' ausdruck
          |  ausdruck '/' ausdruck
          |  '-' ausdruck %prec '*'
          |  NAME
          ;
```

Die Eingabezeile

```
a = b = c*d - e - f*g
```

wird vom Parser so strukturiert:

```
a = ( b = (((c*d) - e) - (f*g)))
```

Zusammenfassung der Regeln, nach denen der Parser Konflikte löst

- Zuerst merkt sich der Parser die Angaben (im Deklarationsteil) über Vorrang und Assoziativität von Token und Einzelzeichen.
- Der Parser ordnet jeder Grammatikregel einen Vorrang und eine Assoziativität zu. Normalerweise nimmt der Parser dafür den Vorrang und die Assoziativität des letzten Tokennamens oder Einzelzeichens, das in der Struktur vorkommt. Falls "%prec TOKEN" in der Grammatikregel angegeben ist, dann werden Vorrang und Assoziativität von *TOKEN* genommen. Es ist auch möglich, daß einigen Grammatikregeln kein Vorrang und keine Assoziativität zugeordnet wird.
- Tritt ein reduziere/reduziere-Konflikt oder ein lies/reduziere-Konflikt auf, bei dem entweder das nächste Eingabesymbol (für "lies") oder die Grammatikregel (für "reduziere") keinen Vorrang oder keine Assoziativität haben, dann arbeitet der Parser nach den Regeln (siehe "Wie werden mehrdeutige Grammatiken und Konflikte behandelt?"):

  – "lies" vor "reduziere",
  – "reduziere" nach der Regel, die zuerst angegeben ist.

  yacc berichtet von Konflikten dieser Art.
- Wenn ein lies/reduziere-Konflikt auftritt und sowohl die Grammatikregel ("reduziere") als auch das Eingabezeichen ("lies") einen Vorrang und eine Assoziativität haben, dann arbeitet der Parser nach den Regeln:

  – Der Parser führt die Aktion(P) ("lies" oder "reduziere") mit der höchsten Priorität aus.
  – Bei gleichem Vorrang führt der Parser je nach Art der Assoziativität folgende Aktion(P) aus:

bei Links-Assoziativität	"reduziere"
bei Rechts-Assoziativität	"lies"
bei Nicht-Assoziativität	"Fehler"

  yacc meldet nicht Konflikte dieser Art.
- Konflikte, die der Parser auch mit obigen Regeln nicht lösen kann, können z.B. auftreten, wenn

  – Fehler in der Eingabe vorliegen,
  – logische Fehler in den Grammatikregeln vorkommen,
  – ein Parser benötigt wird, den yacc nicht generieren kann,
  – Benutzer-Aktionen ausgeführt werden müssen bevor klar ist, welche Regel anzuwenden ist.

Fehlerbehandlung

Die Fehlerbehandlung kann sehr große, insbesondere semantische Probleme bereiten. Wenn ein Fehler gefunden wird, ist es möglicherweise nötig

- Strukturbäume neu aufzubauen, die beschreiben, wie der Parser die Eingabe strukturiert,
- Einträge in Symboltabellen zu löschen oder zu ändern,
- Schalter zu setzen, die verhindern, daß der Parser weitere Ausgabe generiert.

Der Parser versucht trotz der Schwierigkeiten, die Eingabe weiter einzulesen und zu bearbeiten. Der Parser muß dazu einen Punkt in der Eingabe finden, an dem er mit seiner Arbeit fortfahren kann. Wenn keine Regeln zur Fehlerbehandlung angegeben sind, bricht der Parser beim ersten Fehler ab.

Wenn ein Fehler auftritt, passiert folgendes:

- yyerror wird aufgerufen.
- Der Parser geht in den Fehlermodus über.
- Der Parser leert seinen Zustandskeller bis er in einem Zustand ist, in dem das Token error als Eingabe möglich ist.
- Der Parser führt die Aktionen aus, die bei einer passenden Regel mit dem Token error angegeben sind oder bricht ab, wenn keine solche Regel existiert.
- Der Parser bleibt solange im Fehlermodus bis er drei korrekte Token in der Eingabe gefunden hat. Als erstes prüft er das Token, das den Fehler verursacht hat. Im Fehlermodus meldet der Parser keine fehlerhaften Token, sondern ignoriert sie einfach.
- Wenn der Parser drei korrekte Token gefunden hat, geht er in den Normalmodus über.
- Der Parser setzt seine Arbeit mit dem ersten der korrekten Token fort.

Sie können zu einem gewissen Grad die Fehlerbehandlung beeinflussen. Folgende Namen, Funktionen und Variablen stehen Ihnen zur Verfügung:

error

error ist der Name eines (Pseudo-)Token, das für die Fehlerbehandlung reserviert ist. Sie können error in Grammatikregeln verwenden, genau wie einen Token. error muß aber nicht als Token deklariert werden. Wenn in einer Struktur ein Fehler vorliegt, geben Sie auf der rechten Seite error an. In der Aktion, die auf error folgt, können Sie z.B. Tabellen zurücksetzen, Strukturbäume korrigieren oder eine gründlichere Fehlerbehandlung aufrufen.

Der Parser behandelt error wie eine Vorausschau und führt die Aktion aus, die bei der Regel mit error angegeben ist.

Wenn Sie keine Regel mit error angeben, dann bricht der Parser nach dem ersten Fehler ab.

yyerror(s)

Wenn ein Fehler auftritt, wird yyerror aufgerufen.
yyerror gibt eine Fehlermeldung aus. Eine einfache Version von yyerror steht in der yacc-Bibliothek. Sie können aber auch eine eigene Version bereitstellen. yyerror erwartet als Parameter s die Adresse einer Zeichenreihe (char *), wie z.B. "Syntaxfehler".

Nach dem Aufruf von yyerror geht der Parser in den Fehlermodus über. Der Parser bleibt solange im Fehlermodus, bis er drei (vermutlich) korrekte Token eingelesen hat. Wenn dabei ein Fehler vorkommt, gibt der Parser keine Fehlermeldung aus.

yychar

yychar ist eine externe Variable vom Typ integer. yychar enthält die Tokennummer der Vorausschau, bei der ein Fehler entdeckt wurde. Der Fehler ist damit leichter zu finden.

yydebug

yydebug ist eine externe Variable vom Typ integer, die normalerweise den Wert 0 hat. Wenn sie einen Wert ungleich 0 hat, dann gibt der Parser eine ausführliche Beschreibung seiner Aktionen(P) und Eingabezeichen.

Wenn Sie yydebug ungleich 0 setzen wollen, müssen Sie im Deklarationsteil des yacc-Quell-Programms angeben:

```
%{
#define YYDEBUG
%}
```

Innerhalb der Fehlerbehandlung oder als Aktion bei einer Regel, können Sie dann yydebug auf einen Wert ungleich 0 setzen, z.B. mit

```
{ yydebug = 1; }
```

yyerrok;

yyerrok ist eine Anweisung, die der Benutzer im Aktionsteil einer Regel angeben kann. yyerrok veranlaßt, daß der Parser wieder zum Normalmodus zurückkehrt.

Beim Auftreten eines Fehlers geht der Parser normalerweise in den Fehlermodus über. Er bleibt solange im Fehlermodus, bis er drei (vermutlich) korrekte Token eingelesen hat. Wenn dabei ein Fehler vorkommt, überliest der Parser die Token und gibt keine Fehlermeldung aus. Sollen auch Fehlermeldungen darüber ausgegeben werden, müssen Sie yyerrok aufrufen.

yyclearin;

yyclearin ist eine Anweisung, die im Aktionsteil einer Regel stehen kann. Normalerweise wird nach einem Fehler und dem Token error als nächste Vorausschau das Token genommen, bei dem der Fehler auftrat. Wenn dieses Token nicht mehr als Vorausschau verwendet werden soll, müssen Sie yyclearin angeben.

Wenn Sie z.B. in einer Aktion oder einer Funktion zur Fehlerbehandlung festlegen, ab welcher Stelle korrekt im Eingabestrom fortgefahren werden soll, dann ist es sinnvoll mit yyclearin die vorhandene Vorausschau zu löschen.

Beispiele

1. Beispiel:

   Eine Regel der Form

```
anweisung : error
          ;
```

   bewirkt, daß der Parser nach dem Auftreten eines Fehlers, drei Token sucht, die legal auf *anweisung* folgen könnten. Der Parser setzt seine Arbeit bei dem ersten dieser drei Token fort. Falls allerdings nicht klar ist, wo eine Anweisung beendet ist und eine neue anfängt, kann es zu falschen weiteren Fehlermeldungen kommen.

2. Beispiel:

   Eine Regel der Form

```
anweisung : error ';'
                    {
                     ... Fehleraktion ...
                    }
          ;
```

   veranlaßt den Parser die Eingabe bis zu einem Strichpunkt zu überlesen und zu ignorieren.

3. Beispiel:

   Sobald ein Fehler auftritt, wollen Sie die letzte Zeile interaktiv noch einmal eingeben. Da der Parser nach dem Fehler im Fehlermodus ist, meldet er z.B. keine Fehler, die am Anfang der nochmals eingegebenen Zeile aüftreten. Er ignoriert einfach die unkorrekten Token. Um dies zu vermeiden, muß der Parser in den Normalmodus zurückkehren, bevor Sie die Zeile nochmals eingeben.

   Eine Regel dafür lautet:

```
eingabe  :  error '\n'
                    { yyerrok;
                      printf ("Letzte Zeile nochmals eingeben :\n");
                    }
                    eingabe
                    { $$ = $4; }
         ;
```

4. Beispiel:

   Sie stellen eine intelligente Resynchronisationsfunktion *resynch* zur Verfügung, die u.a. beim Auftreten eines Fehlers den Eingabestrom bis zur nächsten korrekten Eingabe weiterliest. Nach einem Aufruf von *resynch* ist dann vermutlich das nächste Token, das yylex zurückgibt, das erste Token in einer erlaubten Struktur. Dieses Token soll die Vorausschau sein und nicht das illegale Token, bei dem der Fehler aufgetreten ist. Der Parser soll aus dem Fehlermodus in den Normalmodus zurückgesetzt werden.

   Eine Regel, die dies ausführt, ist:

```
anweisung  :  error
                    { resynch();
                      yyerrok;
                      yyclearin;
                    }
           ;
```

Welche Möglichkeit gibt es Kontextabhängigkeit zu realisieren?

Der Parser kann globale Variablen mit bestimmten Werten belegen. Wenn z.B. der Scanner einen Eingabetext in Abhängigkeit von einer gefundenen Struktur behandeln soll, dann kann der Parser eine globale Variable belegen. Der Wert der Variablen gibt an, welche Struktur der Parser erkannt hat. Der Scanner fragt den Wert der Variablen ab und handelt entsprechend.

Beispiel

Je nachdem, ob der Parser die Struktur *rede* (mit Anführungsstrichen davor und danach) oder die Struktur *sonst* bearbeitet, soll der Scanner Leerzeichen melden oder ignorieren. Eine Lösung des Problems ist z.B.:

```
%{
  int scannersignal;
%}
  ... Deklarationen ...

% %
text   : sonst rede sonst
       ...
       ;

sonst  : ...
               { scannersignal = 1; }
       ...
       ;

rede   : ...
               { scannersignal = 0; }
       ...
       ;
```

scannersignal steht als globale Variable sowohl dem Scanner als auch dem Parser zur Verfügung.

yacc für Fortgeschrittene

1. Zugriff auf Werte der linken Seite von Regeln

Sie können innerhalb einer Regel auch auf Werte zugreifen, die beim Abarbeiten einer früheren Regel aufgetreten sind. Sie können die Werte mit den Pseudovariablen $0, $-1, $-2, ... ansprechen.

Beispiel

Gegeben sind folgende Regeln eines yacc-Quell-Programms:

```
satz      : artikel adjektiv substantiv verb adjektiv substantiv
                              {...einen Satz betrachten...}
          ;
artikel   : DER               { $$ = DER }
          ;
adjektiv  : UNMUSIKALISCHE    { $$ = UNMUSIKALISCHE }
          ...
          ;
substantiv : MOZART           { if ( $0 == UNMUSIKALISCHE ) {
                                     printf( "Wieso unmusikalisch?\n" );
                                     }
                                $$ = MOZART;
                              }
          ...
          ;
```

Wenn das Token *MOZART* (in der Struktur *substantiv*) eingelesen wird, soll geprüft werden, ob das Adjektiv davor *UNMUSIKALISCHE* war. Um den Wert von *adjektiv* zu erfahren, muß in der Struktur *satz* ein Wert links von *substantiv* abgefragt werden. $0 hat den Wert von *adjektiv*. Den Wert von *artikel* können Sie mit der Pseudovariablen $-1 abfragen.

2. Simulieren der Aktionen(P) "akzeptiere" und "Fehler"

Sie können innerhalb von Aktionen im yacc-Quell-Programm die Aktionen(P) "akzeptiere" bzw. "Fehler" simulieren mit den Makros YYACCEPT bzw. YYERROR.
YYACCEPT veranlaßt, daß yyparse den return-Wert 0 zurückgibt. YYERROR veranlaßt den Parser das aktuelle Eingabesymbol als Fehler zu behandeln. Der Parser hat so die Möglichkeit verschiedene Zeichen als "Ende der Eingabe" zu betrachten bzw. kontext-sensitive Syntax zu überprüfen.

3. Beliebige Datentypen für Werte

Normalerweise sind die Werte des Wertekellers vom Typ integer. Die Werte können aber auch von einem anderen Datentyp sein. Die Wertekellerelemente sind dann Varianten, die alle deklarierten Wertetypen annehmen können.

Sie vereinbaren die Werte als Variante und verknüpfen die Namen der Variantenkomponenten mit den Namen der Token und der syntaktischen Variablen, die entsprechende Werte haben sollen. yacc merkt sich die Typdefinitionen und überprüft die Typen des Parsers genau. Wenn auf einen Wert mit $$, $1, ..., $n zugegriffen wird, setzt yacc automatisch die passende Variantenkomponente ein, so daß keine unerwünschten Typumwandlungen auftreten.

Um Wertetypen einzuführen, müssen Sie folgendes beachten:

- Sie legen mit einer union-Deklaration die Wertetypen fest.
  Für die union-Deklaration haben Sie zwei Möglichkeiten:

  – Sie können im Deklarationsteil des yacc-Quell-Programms mit

    ```
    %union {
            unionkomponente...
            }
    ```

    die Typen des Wertekellers (also von yylval und yyval) festlegen.

    unionkomponente ist eine Komponentenvereinbarung, wie sie normalerweise in einem C-Quell-Programm steht, also z.B.

    ```
    double value;
    ```

  – Sie können in einer Definitionsdatei *defdatei* mit einer typedef-Vereinbarung die Variable YYSTYPE definieren. In *defdatei* steht dann:

    ```
    typedef union {
            unionkomponente...
            } YYSTYPE;
    ```

    Die Definitionsdatei muß im Deklarationsteil des yacc-Quell-Programms eingefügt werden mit:

    ```
    %{
    #include "defdatei"
    %}
    ```

  defdatei ist der Name der Definitionsdatei.

  Die union-Deklaration steht in der Datei y.tab.h, wenn Sie yacc mit Schalter d aufrufen.

- Sie verknüpfen den Namen eines Token oder einer syntaktischen Variablen mit dem Namen der Variantenkomponente, deren Typ es bzw. sie haben soll.

  Der Name einer Variantenkomponente ist z.B. *value*. Dann

  – hat das Token *name* denselben Typ wie *value*, wenn Sie *name* mit

    %**token**␣ < value > ␣name oder
    %**left**␣ < value > ␣name oder
    %**right**␣ < value > ␣name oder
    %**nonassoc**␣ < value > ␣name

    im Deklarationsteil definieren.

  – hat die syntaktische Variable *symbol* denselben Typ wie *value*, wenn Sie im Deklarationsteil

    %**type**␣ < value > ␣symbol

    angeben.

  Als Trennzeichen "␣" können ein oder mehrere Leerzeichen stehen.

- Sie geben den Typ von Werten explizit an, bei denen yacc Schwierigkeiten hat, den Typ zu bestimmen.

  Wenn die Pseudovariablen $$, $1, ..., $n innerhalb einer Aktion von einem bestimmten Typ sein sollen, so versagen eventuell die genannten Möglichkeiten. Wenn Sie die Pseudovariablen innerhalb der Aktion mit

  ```
  $<value>$
  $<value>1  ...
  $<value>n
  ```

  ansprechen, dann sind $$, $1, ... bzw. $n vom selben Typ wie die Variantenkomponente *value*.

- Wenn Sie keine Typangaben machen, geht yacc davon aus, daß die Werte vom Typ integer sind.

Beispiele

1. Beispiel:

   Die Werte sollen vom Typ integer oder double sein. Im Deklarationsteil eines yacc-Quell-Programms steht:

```
%start  rechnung

%union  {
        int ganzezahl;
        double gleitpunktzahl;
        }

%token  <ganzezahl>  RECHNUNGSNUMMER
%token  <gleitpunktzahl>  EINZELBETRAG

%type   <gleitpunktzahl>  endbetrag
```

2. Beispiel:

   Im Regelteil eines yacc-Quell-Programms ist folgende Regel möglich:

```
regel  :  aaa
              { $<ganzezahl>$ = 7; }
          bbb
              { hypot( $<gleitpunktzahl>2, $<gleitpunktzahl>0 ); }
              ;
```

   wenn im Deklarationsteil dieselbe union-Deklaration wie im 1. Beispiel steht.

Hinweis

- Sie sollten Rechtsrekursionen in Regeln vermeiden, da die Keller eventuell zu groß werden und keine Elemente mehr aufnehmen können. Anstatt

```
struktur : NAME
         | NAME struktur
         ;
```

sollten Sie linksrekursive Regeln der Form

```
struktur : NAME
         | struktur NAME
         ;
```

verwenden.

- Oft ist es sinnvoll, leere Zeichenfolgen als erlaubte Struktur anzugeben:

```
struktur : /* leer */
         | struktur NAME
         ;
```

yacc kann allerdings Schwierigkeiten haben, wenn verschiedene Strukturen leer sein können und yacc nicht eindeutig entscheiden kann, welche leere Struktur gemeint ist.

- Die Namen der Dateien, die yacc erstellt und die zusammen übersetzt und gebunden werden, um ein lauffähiges Programm zu erzeugen, sind fest vorgegeben. In einem Dateiverzeichnis können daher niemals zwei oder mehr yacc-Prozesse gleichzeitig aktiv sein.

- yacc hat eventuell Schwierigkeiten aus einem yacc-Quell-Programm einen Parser zu generieren, wenn z.B.

  – die Regeln sich selbst widersprechen,

  – die Eingabesprache zu einer anderen Klasse von Sprachen gehört als die, die yacc behandeln kann.

- Ein Parser, der aus eindeutigen Grammatikregeln generiert wird, arbeitet langsamer und ist schwerer zu schreiben als ein Parser, der aus mehrdeutigen Grammatikregeln erstellt wird. Wie der Parser sich bei mehrdeutigen Regeln verhält, steht unter "Wie werden mehrdeutige Grammatiken und Konflikte behandelt?".

Fehlerdiagnose

- Auf der Standard-Ausgabe wird die Anzahl der reduziere/reduziere- und lies/reduziere-Konflikte angegeben. Detailliertere Angaben stehen in der Datei y.output, die erstellt wird, wenn der Schalter v eingeschaltet ist.
- yacc meldet auch, wenn einige Regeln nicht vom Startsymbol aus erreicht werden können.

Version 2.0

Die Beschreibung des Kommandos yacc ist für alle Versionen (1.0B, 1.0C, 2.0) gültig.

Beispiele

1. Ein einfaches Beispiel:

   Ein Eingabestrom soll nach folgenden Kriterien untersucht werden:

   Anzahl der Zeilen,
   der Worte,
   der Zahlen,
   der Operatoren,
   der sonstigen Zeichen.

   Ein yacc-Quell-Programm für diese Aufgabe steht in der Datei *grammatik*:

```
%{
#include <stdio.h>
long lcounter ;
long wcounter ;
long ncounter ;
long scounter ;
long opcounter ;
%}

%token WORT ZAHL 301 WSPACE 302 OPERATOR NEWLINE OTHER

%%

text          :  /* leer */
              |  text zeile zeilenende
```

```
                ;

zeile           :  /* leer */
                                {
                                  printf("Zeile %ld: ", ++lcounter);
                                }
                |  zeile WORT
                                {
                                  printf("%s", yytext);
                                  ++wcounter;
                                }
                |  zeile ZAHL
                                {
                                  printf("%s", yytext);
                                  ++ncounter;
                                }
                |  zeile OPERATOR
                                {
                                  printf("%s", yytext);
                                  ++opcounter;
                                }
                |  zeile WSPACE
                                {
                                  printf("%s", yytext);
                                }
                |  zeile OTHER
                                {
                                  printf("%s", yytext);
                                  ++scounter;
                                }
                ;

zeilenende      :  NEWLINE
                                {
                                   printf("%s", yytext);
                                }
                ;
% %
#include "lex.yy.c"
```

Das lex-Quell-Programm in der Datei *regeln* lautet:

```
% %
[a-zA-Z][a-zA-Z0-9]*       return (WORT);
[0-9]+                     return (ZAHL);
[ \t]+                     return (WSPACE);
[-+*/%=]                   return (OPERATOR);
[\n]                       return (NEWLINE);
[^-+*/%=0-9a-zA-Z\t \n]+   return (OTHER);
% %
yywrap()
{
    printf("Ihr Text hat %ld Zeile(n)\n",lcounter);
    printf("             %ld Wort(e)\n",wcounter);
    printf("             %ld Zahl(en)\n",ncounter);
    printf("             %ld Operator(en)\n",opcounter);
    printf("             %ld sonstige(s) Zeichen\n",scounter);
    return(1);
}
```

In der Datei *text* steht folgender Text:

```
Ein Moench aus Tik-se,
der hatte 'ne Schickse.
Die 2, die gingen nach She-i,
da waren's bald d'rauf 3.
```

Die Kommandos

```
$ yacc grammatik
$ lex regeln
$ cc y.tab.c -ly -ll
$ a.out < text
```

führen zu der Ausgabe

```
Zeile 1: Ein Moench aus Tik-se,
Zeile 2: der hatte 'ne Schickse.
Zeile 3: Die 2, die gingen nach She-i,
Zeile 4: da waren's bald d'rauf 3.
Ihr Text hat 4 Zeile(n)
             21 Wort(e)
             2 Zahl(en)
             2 Operator(en)
             8 sonstige(s) Zeichen
$
```

2. Ein Tischrechner

Als nächstes soll ein einfacher Tischrechner realisiert werden. Er hat 26 Register, die "a", "b",..., "z" heißen. Der Tischrechner kann die Operationen "+", "-", "*", "/", "%", "&" (bit-weise Konjunktion), "|" (bit-weise Disjunktion) und Zuweisung ausführen. Er kann ganze Zahlen behandeln, die Sie auch oktal angeben können (mit 0 davor).

Das Beispiel zeigt, wie Präzedenzen behandelt werden und eine einfache Fehlerbehandlung erfolgt. Das Ergebnis von Ausdrücken wird auf der Standard-Ausgabe aufgezeigt. Nur Anweisungen werden nicht ausgedruckt. Die lexikalische Analyse ist sehr einfach realisiert.

In der Datei *grammatik* steht folgendes yacc-Quell-Programm:

```
%{
#include <stdio.h>
#include <ctype.h>

int rregister[26];
int basis;
%}

%start liste

%token ZIFFER BUCHSTABE

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left EMINUS

% %                              /* Regeln */

liste      : /* leer */
           | liste  anweisung '\n'
           | liste error '\n'
                 { yyerrok; }
           ;

anweisung  :  ausdruck
                 { printf("%d\n", $1); }
           |  BUCHSTABE '=' ausdruck
                 { rregister[$1] = $3; }
           ;
```

```
ausdruck   :  '(' ausdruck ')'
                   { $$ = $2; }
           |  ausdruck '+' ausdruck
                   { $$ = $1 + $3; }
           |  ausdruck '-' ausdruck
                   { $$ = $1 - $3; }
           |  ausdruck '*' ausdruck
                   { $$ = $1 * $3; }
           |  ausdruck '/' ausdruck
                   { $$ = $1 / $3; }
           |  ausdruck '%' ausdruck
                   { $$ = $1 % $3; }
           |  ausdruck '&' ausdruck
                   { $$ = $1 & $3; }
           |  ausdruck '|' ausdruck
                   { $$ = $1 | $3; }
           |  '-' ausdruck %prec EMINUS
                   { $$ = - $2; }
           |  BUCHSTABE
                   { $$ = rregister[$1]; }
           |  zahl
           ;

zahl       :  ZIFFER
                   { $$ = $1; basis = ($1 == 0) ? 8 : 10; }
           |  zahl ZIFFER
                   { $$ = basis * $1 + $2; }
           ;

%%

yylex()    {

             int c;

             while ((c = getchar()) == ' ')
                     ;
             if ( islower (c)) {
                  yylval = c - 'a';
                  return (BUCHSTABE);
                  }
             if ( isdigit (c)) {
                  yylval = c - '0';
                  return (ZIFFER);
                  }
             return (c);
           }
```

Nach den Kommandos

```
$ yacc grammatik
$ cc y.tab.c -ly
```

steht der ablauffähige Tischrechner in der Datei a.out.

```
$a.out ↵
a = 8*9 - (6-7) ↵
a ↵
73
5+7 - 8*9 ↵
-60
b = 6 ↵
c = a+b ↵
c ↵
79
f = 0345 ↵
f ↵
229 END
$
```

3. Erweiterter Tischrechner

   Das nächste Beispiel soll dem fortgeschrittenen yacc-Benutzer weitere Möglichkeiten von yacc zeigen.

   Der Tischrechner kann mit Gleitpunktkonstanten und Gleitpunktintervallen rechnen. Er kennt die Operationen "+", "-", "*", "/" und "=". Weiter arbeitet er mit 26 Gleitpunktvariablen ("a", "b", ..., "z") und 26 Intervallvariablen ("A", "B", ..., "Z"). Ein Intervall muß als

   ```
   (anfang, ende)
   ```

   angegeben werden. *anfang* und *ende* müssen Gleitpunktkonstanten oder Gleitpunktvariablen sein; *anfang* muß kleiner oder gleich *ende* sein. Eine Gleitpunktkonstante kann wie in C geschrieben werden, allerdings nicht mit "E", sondern nur mit "e" (z.B. -3.5e9). Die Werte von Ausdrücken werden auf der Standard-Ausgabe ausgegeben. Nur bei Zuweisungen ("=") unterbleibt die Ausgabe.

   Intervalle sind als Strukturen definiert mit Komponenten vom Typ double. Der Wertekeller des Parsers kann Elemente vom Typ integer (die Variablen), vom Typ double (die Gleitpunktzahlen) oder vom Typ INTERVAL (die Intervalle) haben. Ein Gleitpunktausdruck kann auch als Intervallausdruck verwendet werden.

   Ein Fehler, der mit YYERROR behandelt wird, ist

   - ein Intervall, bei dem *anfang* größer als *ende* ist,
   - Division durch ein Intervall, das 0 enthält.

   Wenn ein Fehler auftritt, ignoriert der Parser den Rest der Zeile.

   Der Scanner kann Variablen (*VREGISTER*, *DREGISTER*), Gleitpunktzahlen (*KONSTANTE*) oder Einzelzeichen (z.B. '.', 'e') als return-Werte zurückgeben. Mit den Variablen yylval.*ival* bzw. yylval.*dval* übergibt der Scanner die Werte der Token *VREGISTER*, *DREGISTER* bzw. *KONSTANTE* an den Parser. Die Funktion atof wandelt eine Zeichenreihe in eine Gleitpunktzahl um. Wenn der Scanner einen Fehler in der Eingabe bemerkt (z.B. mehr als einen Punkt oder ein "e" in einer Gleitpunktzahl), dann gibt er das falsche Zeichen an den Parser und dieser übernimmt die Fehlerbehandlung.

Das Hauptproblem liegt im Unterscheiden von Klammerausdrücken der Form

2.5 + (3.5 – 4.)

und

2.5 + (3.5 , 4.)

Bis der Parser das Komma im zweiten Ausdruck eingelesen hat, weiß er nicht, daß er "2.5" in ein Intervall umwandeln muß.

Allgemein müssen oft mehrere Token eingelesen werden, bevor der Parser weiß, welche Struktur er vor sich hat. Das Problem wird gelöst mit zwei Regeln für jeden zweistelligen Operator. Eine Regel behandelt den Fall, wenn der linke Operand kein Intervall, die andere den Fall, wenn der linke Operand ein Intervall ist. Im letzten Fall wird der rechte Operand immer in ein Intervall umgewandelt. Die Regeln für Intervallausdrücke stehen nach denen für Gleitpunktausdrücke, so daß eine Umwandlung erst dann erfolgt, wenn es wirklich nötig ist.

Eine bessere, aber weniger lehrreiche Lösung des Problems wäre es, die Information über den Tokentyp als Tokenwert und nicht als Teil der Grammatik zu behandeln.

In der Datei *grammatik* steht das folgende yacc-Quell-Programm:

```
%{

 /* Definitionen und Anweisungen für die Aktionen im
    yacc-Quell-Programm */

#include <stdio.h>
#include <ctype.h>

typedef struct interval {
        double un, ob;
        }   INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dregister[26];
INTERVAL vregister[26];
%}
```

```
 /* Deklarationen und Definitionen für yacc */

%start zeilen                       /* Startsymbol */

%union {                            /* Typdeklaration für yylval und
                                       yyval */

       int ival;
       double dval;
       INTERVAL vval;
       }

%token <ival> DREGISTER VREGISTER   /* Token für die Register */

%token <dval> KONSTANTE             /* Token für Gleitpunktkonstante */

%type <dval> dausdruck              /* Typ der syntaktischen Variablen für
                                             Gleitpunktzahlen */

%type <vval> vausdruck              /* Typ der syntaktischen Variablen für
                                       Gleitpunktintervalle */

%left '+' '-'                       /* Vorrang und Assoziativität */
%left '*' '/'
%left EMINUS                        /* Vorrang für einstelliges Minus */

%%

 /* Regeln im yacc-Quell-Programm */

zeilen     : /* leer */             /* Startsymbol */
           | zeilen zeile
           ;

zeile      :  dausdruck '\n'
                   { printf("%15.8f\n", $1); }
           |  vausdruck '\n'
                   { printf("(%15.8f, %15.8f)\n", $1.un, $1.ob); }
           |  DREGISTER '=' dausdruck '\n'
                   { dregister[$1] = $3; }
           |  VREGISTER '=' vausdruck '\n'
                   { vregister[$1] = $3; }
           |  error '\n'
                   { yyerrok; }
           ;
```

```
dausdruck  :  KONSTANTE              /* Gleitpunktzahl */
           |  DREGISTER
                { $$ = dregister[$1]; }
           |  dausdruck '+' dausdruck
                { $$ = $1 + $3; }
           |  dausdruck '-' dausdruck
                { $$ = $1 - $3; }
           |  dausdruck '*' dausdruck
                { $$ = $1 * $3; }
           |  dausdruck '/' dausdruck
                { $$ = $1 / $3; }
           |  '-' dausdruck %prec EMINUS
                { $$ = - $2; }
           |  '(' dausdruck ')'
                { $$ = $2; }
           ;

vausdruck  :  dausdruck              /* Intervall */
                { $$.un = $$.ob = $1; }
           |  '(' dausdruck ',' dausdruck ')'
                { $$.un = $2;
                  $$.ob = $4;
                  if ( $$.un > $$.ob ) {
                       printf("Intervall nicht korrekt\n");
                       YYERROR;
                       }
                }
           |  VREGISTER
                { $$ = vregister[$1]; }
           |  vausdruck '+' vausdruck
                { $$.ob = $1.ob + $3.ob;
                  $$.un = $1.un + $3.un;
                }
           |  dausdruck '+' vausdruck
                { $$.ob = $1 + $3.ob;
                  $$.un = $1 + $3.un;
                }
           |  vausdruck '-' vausdruck
                { $$.ob = $1.ob - $3.un;
                  $$.un = $1.un - $3.ob;
                }
           |  dausdruck '-' vausdruck
                { $$.ob = $1 - $3.un;
                  $$.un = $1 - $3.ob;
                }
           |  vausdruck '*' vausdruck
                { $$ = vmul( $1.un, $1.ob, $3 ); }
```

```
        | dausdruck '*' vausdruck
              { $$ = vmul( $1, $1, $3 ); }
        | vausdruck '/' vausdruck
              { if ( dcheck($3) )
                    YYERROR;
                $$ = vdiv($1.un, $1.ob, $3 );
              }
        | dausdruck '/' vausdruck
              { if ( dcheck($3) )
                    YYERROR;
                $$ = vdiv($1, $1, $3 );
              }
        | '-' vausdruck %prec EMINUS
              { $$.ob = - $2.un;
                $$.un = - $2.ob;
              }
        | '(' vausdruck ')'
              { $$ = $2; }
        ;

%%

 /* Scanner und Programme für die Aktionen */

#define BSZ 50                      /* Puffergröße für
                                       Gleitpunktzahlen */

yylex()    {                        /* Scanner */

            int c;

            while ((c = getchar()) == ' ')
                ;
            if ( isupper (c)) {   /* c Intervall-Register */
                yylval.ival = c - 'A';
                return (VREGISTER);
                }
            if ( islower (c)) {   /* c Gleitpunkt-Register */
                yylval.ival = c - 'a';
                return (DREGISTER);
                }
            if ( isdigit (c) || c == '.') {
                                  /* c Gleitpunktkonstante oder Fehler */
                char buf[BSZ+1],  *cp = buf;
                int dot = 0, exp = 0;
```

```
                for ( ; (cp-buf) < BSZ ; ++cp, c = getchar() ) {

                        *cp = c;
                        if ( isdigit(c) )
                             continue;
                        if ( c == '.' ) {
                             if ( dot++ || exp) /* Fehler */
                                  return('.');
                             continue;
                             }

                        if ( c == 'e' ) {
                             if (exp++) /* Fehler */
                                 return('e');
                             continue;
                             }
                        /* Ende der Zahl erreicht */
                        break;
                        }

                    *cp = '\0';
                    if ((cp-buf) >= BSZ)
                         printf ("Konstante zu lang\n");
                    else
                         /* letztes gelesenes Zeichen wieder
                            zurückstellen */
                        ungetc(c,stdin);
                    yylval.dval = atof (buf);
                    return (KONSTANTE);
                    }
              return(c);
            }

INTERVAL obun (a, b, c, d)          /* obun berechnet das kleinste
                                       Intervall, das a, b, c und d
                                       enthält */
double a, b, c, d;
             {
               INTERVAL v;

               if (a>b) {
                   v.ob = a; v.un = b;
                   }
               else {
                    v.ob = b; v.un = a;
                    }
```

```
            if (c>d) {
                if ( c > v.ob ) v.ob = c;
                if ( d < v.un ) v.un = d;
                }
            else {
                if ( d > v.ob ) v.ob = d;
                if ( c < v.un ) v.un = c;
                }
            return (v) ;
          }

INTERVAL vmul (a, b, v)              /* Intervall-Multiplikation */
double a, b; INTERVAL v;
          {
          return ( obun (a*v.ob, a*v.un, b*v.ob, b*v.un ));
          }

dcheck (v)                           /* 0 liegt im Intervall v */
INTERVAL v;
          {
          if ( v.ob >= 0. && v.un <= 0. ) {
               printf("0 liegt im Divisorintervall.\n");
               return(1);
               }
          return(0);
          }

INTERVAL vdiv (a,b,v)                /* Intervall-Division */
double a, b;
INTERVAL v;
          {
          return ( obun (a/v.ob, a/v.un, b/v.ob, b/v.un ));
          }

$
Konflikte: 18 lies/reduziere, 26 reduziere/reduziere
$
```

yacc meldet 18 lies/reduziere- und 26 reduziere/reduziere-Konflikte.

Nach den Kommandos

```
$
$
```

können Sie die Ausdrücke, die der Tischrechner berechnen soll, auf der Standard-Eingabe eingeben.

Dateien

y.output	enthält eine Beschreibung der Parsing-Tabellen und der Konflikte (siehe Schalter v)
y.tab.c	enthält yyparse
y.tab.h	enthält Definitionen von Tokennummern für die Token (siehe Schalter d)
yacc.tmp, yacc.acts	Zwischendateien
/usr/lib/yaccpar	enthält einen Parser Prototyp für C-Programme
/lib/liby.a	yacc-Bibliothek, mit einer einfachen Standard-Version von "main" und "yyerror"

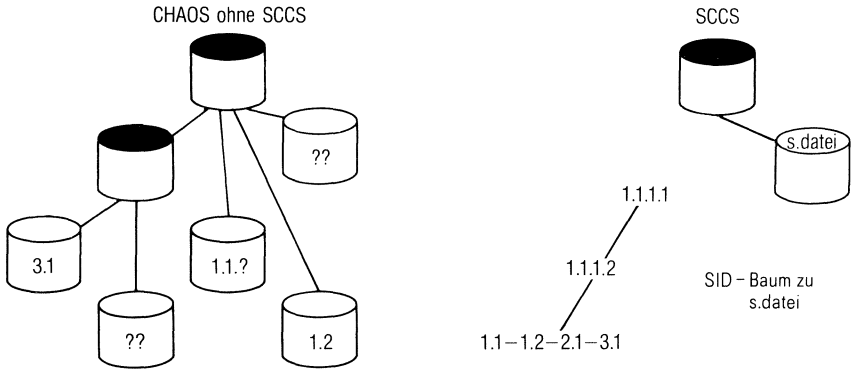
Siehe auch

A.V.Aho and S.C.Johnson,
LR Parsing,
Computing Surveys, Juni 1978

S.C.Johnson,
YACC - Yet Another Compiler Compiler

> > > > lex

3 SCCS - Source Code Control System



3.1 SCCS allgemein

Was ist SCCS?

SCCS (Source Code Control System) ist ein System zur Verwaltung und Entwicklung von Textdateien, z.B. Dateien mit Quellcode oder Dokumentation.

Warum SCCS?

Umfangreiche Texte oder Programmsysteme, die auf viele Dateien verteilt sind, müssen oft (von mehreren Benutzern) erstellt, gewartet, verbessert, ausgeführt und erweitert werden. Insbesondere müssen verschiedene Versionen greifbar sein. SCCS hilft beim Verwalten der Dateien und beim Wiederauffinden von bestimmten Versionen. SCCS spart Speicherplatz dadurch, daß es nur die Dateien mit ihrem ursprünglichen Inhalt und die jeweiligen Veränderungen speichert.

SCCS kann

- verschiedene Versionen einer (Quell)-Datei
  - speichern,
  - generieren,
  - aktualisieren,
  - löschen,
  - ausgeben,
  - vergleichen,
- Zugriffsrechte überprüfen,
- Dateiversionen bestimmen,
- Datum, Urheber und Grund von Änderungen protokollieren,
- verhindern, daß mehrere Benutzer gleichzeitig auf dieselbe Version einer Datei zugreifen.

SCCS soll

- Ordnung schaffen,
- die Arbeit verschiedener Entwickler koordinieren,
- die Wartung von Programmen oder Texten erleichtern,
- die Entwicklung paralleler Versionen ermöglichen.

Vorteile des SCCS

- Ordnung und Überblick in großen Dateisystemen mit verschiedenen Versionen
- Weitgehende Konsistenz
- Speicherplatzersparnis (siehe "Die s-Datei")
- Leichtere Fehlerlokalisierung und Kommunikation durch die Versionsnummer
- Allgemeine Verwendungsmöglichkeiten (z.B. Programme, Dokumente,...)
- "Einfacher Zugriff" auf einzelne Versionen
- Trennung von Entwicklungs- und Auslieferungsversionen
- Dokumentation von Änderungen
- Erleichterung der Projektleitung (z.B. Zugriffsschutz)

Nachteile des SCCS

- Mehr Verwaltungsaufwand
- Zeitverlust beim Ziehen einer Version (alle Vorgängerversionen müssen durchlaufen werden)
- Versionsabhängigkeit (geht eine Zwischenversion verloren, können eventuell die Folgeversionen nicht mehr rekonstruiert werden)
- Zu großes Sicherheitsgefühl (verliert man eine s-Datei, dann sind sämtliche Versionen verloren)
- Baum für jede Datei (kann mit Bibliotheken vermieden werden)
- Riesige Deltas bei globalen Änderungen
- Zusammenführung von Zweigen nicht möglich

3.1.1 Wie funktioniert SCCS?

An einem Beispiel soll der Gebrauch des SCCS gezeigt werden.

Die Datei *tiere* hat den Inhalt

```
hund
katze
vogel
pinguin
hängebauchschwein
```

Mit dem Kommando

```
$
```

übergeben Sie die Datei *tiere* dem SCCS. admin legt eine sogenannte s-Datei an, die alle späteren Versionen der übergebenen Datei enthalten wird. Die Namen von s-Dateien müssen immer mit "s." beginnen. In unserem Fall wird die SCCS-Datei *s.tiere* erstellt.

Sobald sicher ist, daß admin die s-Datei angelegt hat:

Originaldatei entfernen!

Alle Änderungen dürfen nur noch über SCCS stattfinden.
admin antwortet mit der Warnung:

```
No id keywords (cm7)
$
```

"id keywords" sind Identifikations-Schlüsselwörter, die irgendwo im Text einer Version stehen können (siehe get, "Identifikations-Schlüsselwörter). Sie erleichtern das Identifizieren einer Version, z.B. wann die Version erstellt wurde und welche SID-Nummer sie hat. Sie haben das Format

%großbuchstabe%

Je nachdem welcher Buchstabe *großbuchstabe* ist, wird ein Text (z.B. das Datum) an die Stelle von *großbuchstabe* geschrieben, wenn get die Version zum Lesen aus dem SCCS holt:

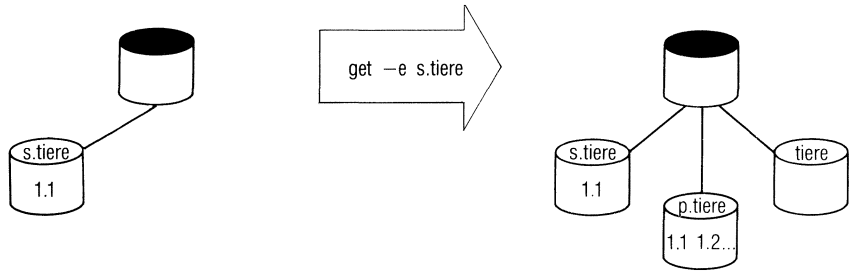
```
$ get s.tiere
1.1
5 lines
No id keywords (cm7)
$
```

get holt die jüngste Version, in unserem Fall 1.1, aus dem SCCS und erstellt die Datei *tiere*. Der Text steht nur zum Lesen in dieser Datei.
Um den Text der SCCS-Datei zu ändern, müssen Sie eingeben:

```
$ get -e s.tiere
1.1
new delta 1.2
5 lines
$
```

get erstellt diesmal eine Datei *tiere*, die Sie lesen und verändern können. Die Datei *tiere* heißt g-Datei. Der Name der g-Datei ist derselbe wie der Name der s-Datei ohne "s." am Anfang.
Ein neues Delta wird die Nummer 1.2 haben. Wenn nichts angegeben ist, erhöht sich die Levelnummer um 1, die Releasenummer bleibt gleich.
Die erste Zahl von links ist die Releasenummer (hier: 1), die zweite die Levelnummer (hier: 2). "Delta" ist eine andere Bezeichnung für "Version".

Zusätzlich errichtet get eine sogenannte p-Datei, die Informationen über die SCCS-Datei enthält. Die Informationen betreffen den Zugriffsschutz, die spätere Wiedereingliederung der Version und sonstige Angaben zur Verwaltung der Version. Der Name der p-Datei ist derselbe wie der der s-Datei außer, daß "s." am Anfang durch "p." ersetzt wird.



Mit einem beliebigen Editor können Sie jetzt den Text der Datei *tiere* verändern, löschen oder erweitern.

Wir hängen ans Ende der Datei *tiere* noch zwei Zeilen an:

```
waschbär
rhinozeros
```

```
$
comments?
```

Mit delta läßt sich die veränderte Version dem SCCS eingliedern.
Zuerst fragt delta nach einem Kommentar. Der Kommentar kann ein beliebiger Text oder leer sein. Sie geben den Kommentar interaktiv ein.
Dann informiert sich delta in der p-Datei, welche Version der Datei *s.tiere* Sie geändert haben. delta holt sich diese Version und vergleicht sie mit der veränderten Version.
Aus der p-Datei erfährt delta auch, welche Nummer die neue Version haben soll. Die neue Versionsnummer und die Unterschiede, die die neue Version gegenüber der Vorgängerversion hat, speichert delta in der s-Datei ab.
delta gibt auf der Standard-Ausgabe aus:

```
No id keywords (cm7)
1.2
2 inserted
0 deleted
5 unchanged
$
```

delta löscht die g-Datei *tiere*. Außerdem löscht delta in der p-Datei den Eintrag über die Version, die Sie gerade erstellt haben. Wenn keine Einträge mehr in der p-Datei stehen, wird auch die p-Datei gelöscht.

Wenn Sie die Release 2 einführen wollen, muß der Aufruf lauten

```
$ get -e -r2 s.tiere
1.2
new delta 2.1
7 lines
$
```

get holt die Version 1.2 zum Ändern. Die neue Version wird die Nummer 2.1 haben.

3.1.2 Die Dateien des SCCS

Die s-Datei

Wenn Sie eine Datei dem SCCS übergeben, wird eine **s-Datei** angelegt. Der Name einer s-Datei muß immer mit "s." beginnen. Wenn von einer "SCCS-Datei" gesprochen wird, ist im allgemeinen immer eine s-Datei gemeint.

Die Originaldatei wird in ihrer Gesamtheit in der s-Datei abgespeichert. Für die nachfolgenden Versionen wird jeweils nur die Differenz zur Vorgängerversion abgespeichert. In der s-Datei stehen sämtliche Versionen - die Originaldatei und alle Änderungen.

Mit **Delta** bezeichnet man

- die Differenz einer Version zu einer Nachfolgerversion,
- *aber auch* allgemein eine Version.

Jede Version (oder jedes Delta) hat einen Namen. Der Name heißt entweder

- die **SID-Nummer** oder
- die **Versionsnummer** oder
- die **Deltanummer**.

SID ist die Abkürzung von "SCCS Identification String".

Der Benutzer bezeichnet eine Version normalerweise mit der Versionsnummer (Ausnahme: get -a). Welche Versionsnummer (Deltanummer, SID-Nummer) eine Version (ein Delta) bekommt, wird im Abschnitt "Die Deltanummer und der SID-Baum" beschrieben.

Zusätzlich zur SID-Nummer bekommt jede Version eine **laufende Nummer**. Die laufenden Nummern geben an, in welcher zeitlichen Reihenfolge die Versionen erstellt wurden. Die laufende Nummer ist eine ganze Zahl. Wenn eine neue Version eingeführt wird, bekommt sie als laufende Nummer die Zahl z+1, wenn z die laufende Nummer war, die als letztes vergeben wurde. Die Ursprungsdatei einer SCCS-Datei hat die laufende Nummer 0, die erste Version hat die Nummer 1.

In der s-Datei stehen für jede Version (außer den Textänderungen) noch

- Informationen über Datum, Autor und Grund der Version,
- Kommentare,
- globale Zusatzinformationen.

Format einer s-Datei

Eine s-Datei besteht aus Zeilen mit ASCII-Text. Der Text ist in 6 Teile gegliedert:

- **Prüfsumme**
  Die erste Zeile enthält die "logische" Summe aller Zeichen, die in der Datei, ohne die erste Zeile, stehen. Die Summe dient der Konsistenzprüfung.

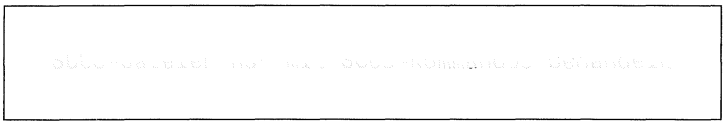
- **Delta-Tabelle**
  Die Tabelle enthält Informationen über jedes Delta, wie z.B. den Typ (siehe rmdel), die SID-Nummer, Tag und Uhrzeit der Erstellung, den Namen des Erstellers, die laufende Nummer des Deltas und des Vorgängerdeltas, die MR-Nummern (siehe admin, Schalter -fv und -m), den Kommentar (siehe delta, Schalter -y).

- **Benutzerliste**
  Die Benutzerliste ist eine Liste mit login-Namen oder/und Gruppennummern der Benutzer, die die SCCS-Datei ändern und Deltas erstellen oder entfernen dürfen (siehe admin, Schalter -a und -e).

- **Parameter**
  Sie können verschiedene Parameter setzen und mit bestimmten Werten belegen (siehe admin, Schalter -f und -d). Die Parameter bestimmen u.a., wie einzelne SCCS-Kommandos auszuführen sind.

- **Beschreibender Text**
  Der Benutzer kann hier eine ausführliche Beschreibung des eigentlichen Textes geben, der in der SCCS-Datei steht.

- **Eigentlicher Text**
  Hier steht der Text, den SCCS in der s-Datei verwaltet. Der Text besteht aus der Ursprungsversion und sämtlichen Änderungen, die Sie bei jeder Version vorgenommen haben. Zwischen den eigentlichen Textzeilen können Kontrollzeilen des SCCS stehen.



Die x-Datei

Alle SCCS-Kommandos, die eine SCCS-Datei verändern, erstellen eine temporäre Kopie der Datei, die **x-Datei** (statt "s." am Anfang "x."), und bearbeiten die x-Datei. Wenn das Kommando fehlerfrei ausgeführt und korrekt beendet wird, bekommt die x-Datei den Namen der s-Datei, d.h. "x" wird durch "s" ersetzt. s-Dateien dürfen deshalb keine Verweise von anderen Dateiverzeichnissen haben (siehe ln, Betriebssystem SINIX, Buch 1). Falls beim Ausführen des Kommandos Fehler auftreten, wird höchstens die x-Datei, nicht aber die s-Datei zerstört.

Die x-Datei steht im selben Dateiverzeichnis wie die s-Datei und hat auch dieselbe Schutzbiteinstellung. Wenn die s-Datei noch nicht existiert (z.B. bei admin), ist die Schutzbiteinstellung `-r--r--r--` bei Version 1.0B, `-r--------` bei Version 1.0C. Die Schutzbiteinstellung einer Version wird noch geändert, wenn Sie ein entsprechendes umask-Kommando eingegeben haben (siehe Betriebssystem SINIX, Buch 1). Der effektive Benutzer ist der Eigentümer der x-Datei.

Der Benutzer kann die Existenz der x-Datei normalerweise ignorieren. Wenn das System zusammenbricht oder sonst etwas Unvorhergesehenes passiert, kann die x-Datei eventuell helfen, den Zustand vor dem Zusammenbruch zu rekonstruieren.

Die z-Datei

Alle SCCS-Kommandos, die eine SCCS-Datei verändern, erstellen eine **z-Datei** (statt "s." am Anfang "z."). Die z-Datei existiert nur während der Ausführung des Kommandos und blockiert weitere Zugriffe auf die SCCS-Datei, solange das Kommando abgearbeitet wird.
In der z-Datei steht die Prozeßnummer des Kommandos, das die z-Dateierstellt hat. Wenn eine z-Datei existiert, erkennen andere SCCS-Kommandos, daß die entsprechende SCCS-Datei blockiert ist.

Die z-Datei hat die Schutzbiteinstellung `-r--r--r--` (Version 1.0B), bzw. `-r--------` (Version 1.0C). Sie steht im selben Dateiverzeichnis wie die s-Datei. Der effektive Benutzer ist der Eigentümer.

Der Benutzer kann die Existenz der z-Datei normalerweise ignorieren. Wenn das System zusammenbricht oder sonst etwas Unvorhergesehenes passiert,kann die z-Datei eventuell helfen, den Zustand vor dem Zusammenbruch zu rekonstruieren.

Die g-Datei

Das Kommando get erstellt zu einer s-Datei die **g-Datei**. Die g-Datei hat denselben Namen wie die s-Datei ohne "s." am Anfang. Beim Kommando get können Sie Genaueres über die g-Datei nachlesen.

Die p-Datei

Das Kommando get erstellt zu einer s-Datei die **p-Datei**. Die p-Datei hat denselben Namen wie die s-Datei, aber "p." statt "s." am Anfang. Beim Kommando get können Sie Genaueres über die p-Datei nachlesen.

Die d-Datei

Das Kommando delta erstellt zu einer s-Datei die **d-Datei**. Die d-Datei hat denselben Namen wie die s-Datei, aber "d." statt "s." am Anfang. Beim Kommando delta können Sie Genaueres über die d-Datei nachlesen.

Die l-Datei

Das Kommando get erstellt zu einer s-Datei eine **l-Datei**, wenn Sie den Schalter -l angeben. Die l-Datei hat denselben Namen wie die s-Datei, aber "l." statt "s." am Anfang. Beim Kommando get können Sie mehr über die l-Datei nachlesen.

Zugriffsschutz für SCCS-Dateien

Wie können Sie Ihre SCCS-Datei schützen?

SCCS stellt einige Schalter und Parameter bei admin zur Verfügung, die den Zugriff auf SCCS-Dateien schützen:

-a, -e	Bei diesen Schaltern können Sie eine Liste der Benutzer angeben, die Deltas erstellen, bzw. nicht erstellen dürfen.
-fc, -ff	Mit diesen Parametern können Sie den Bereich derReleasenummern festlegen.
-fl	Sie können Releasenummern sperren. Versionen mit gesperrten Releasenummern dürfen nicht mehr verändert werden.

Wie schützt SCCS seine Dateien?

- admin gibt neuen SCCSDateien die Schutzbiteinstellung `-r--r--r--`(Version 1.0B), bzw. `-r--------` (Version 1.0C). Sie sollten die Zugriffsrechte nicht ändern, um sicherzustellen, daß Sie die SCCS-Dateien nur mit SCCS-Kommandos bearbeiten.
- Dateiverzeichnisse, die SCCS-Dateien enthalten, sollten die Schutzbiteinstellung drwxr-xr-x haben, damit nur der Eigentümer Einträge ändern darf. Dateiverzeichnisse mit SCCS-Dateien sollten keine Dateien enthalten, die nicht vom SCCS erstellt worden sind.
- Namen von SCCS-Dateien müssen mit "s." anfangen. Die SCCS-Dateien dürfen keine Verweise von anderen Dateiverzeichnissen haben (siehe ln, Betriebssystem SINIX, Buch 1).

Wie können Sie SCCS-Dateien bei mehreren Benutzern schützen?

Wenn mehrere Benutzer mit derselben SCCS-Datei arbeiten, sollte einer zum Eigentümer und Verwalter gewählt werden. Nurder Verwalter darf admin aufrufen.

Ein projektabhängiges (Schnittstellen)-Programm oder eine passende Shell-Prozedur muß den anderen Benutzern Schreiberlaubnis für die Kommandos get, unget, delta, cdc und eventuell rmdel geben. Eigentümer dieses Programms muß der Verwalter sein. Bei dem Programm muß der Verwalter das s-Bit gesetzt haben, damit es unter seiner Benutzernummer abläuft. Die effektive Benutzernummer ist dann die des Verwalters.

Das Schnittstellenprogramm ruft die jeweiligen SCCS-Kommandos auf und vererbt seine effektive Benutzernummer dem Kommando, solange es ausgeführt wird. Benutzer, die nicht Eigentümer einer SCCS-Datei sind, aber in der Benutzerliste der SCCS-Datei stehen (admin, Schalter -a), dürfen nur über das Schnittstellenprogramm die Kommandos delta, get, unget, rmdel und cdc aufrufen.

3.1.3 Die Deltanummer und der SID-Baum

Die Deltanummern (SID-Nummern, Versionsnummern) können als die Knoten des **SID-Baumes** oder **SCCS-Baumes** betrachtet werden. Die Wurzel ist die erste Version. Sie hat z.B. die Nummer 1.1. Nachfolgerdeltas heißen z.B. 1.2, 1.3, ..., 2.1, 2.2, ... Jedes Delta ist von allen früheren Deltas abhängig. Der Baum besteht bis jetzt nur aus dem Stamm und sieht so aus:

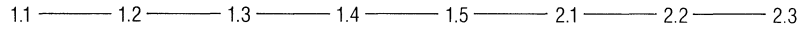


Bild 3-1 Einfacher SID-Baum

Bei einer Deltanummer heißt

- die erste Komponente **Releasenummer** oder **Release**
- die zweite Komponente **Levelnummer** oder **Level**

2.3 hat z.B. die Releasenummer 2 und die Levelnummer 3.

Wenn Sie eine neue Version einführen, wird normalerweise die Levelnummer um 1 hochgezählt, die Release bleibt unverändert. Wenn Sie die Release verändern wollen, müssen Sie es explizit angeben. Wenn z.B. die höchste Releasenummer eines Deltas 4 war und die neueste Version Releasenummer 7 haben soll, dann müssen Sie die SCCS-Datei holen mit

```
$ get -r7 -e s.datei
```

Oft ist es nötig Änderungen vorzunehmen, die nicht von allen früheren Deltas abhängen.

Sie haben z.B. folgendes Programmsystem im SCCS:

Die Programme der Version 1.3 sind freigegeben und können von Kunden benutzt werden. Version 2.1 und 2.2 sind bereits entwickelt, aber noch nicht nicht freigegeben. Sie arbeiten gerade an Version 2.3. Ein Kunde meldet einen Fehler in Version 1.3, den Sie möglichst bald beheben wollen. Sie möchten ein Delta zu Version 1.3 erstellen und für den Kunden freigeben. Dieses Delta soll nicht von einer Version mit Releasenummer 2 abhängen. Der SID-Baum muß einen Zweig bekommen.

Deltanummern von Zweigen bestehen aus vier Komponenten:

- der **Releasenummer (Release)**
- der **Levelnummer (Level)**
- der **Zweignummer (Branch)**
- der **Folgenummer (Sequence)**

Release- und Levelnummer sind die entsprechenden Nummern des Stammknotens, von dem der Zweig gebildet werden soll. Im Beispiel also 1.3. Die Zweignummer gibt an, welcher Zweig zu einem Stammknoten gebildet wird.

Der erste Zweig hat die Zweignummer 1, der nächste die Zweignummer 2usw. Innerhalb eines Zweiges erhält jeder Knoten eine Folgenummer. Die Folgenummer richtet sich danach, an welcher Stelle der Knoten innerhalb des Zweiges liegt. Zweige können auch in anderen Zweigknoten beginnen.

Das Beispiel kann folgenden SID-Baum bekommen:

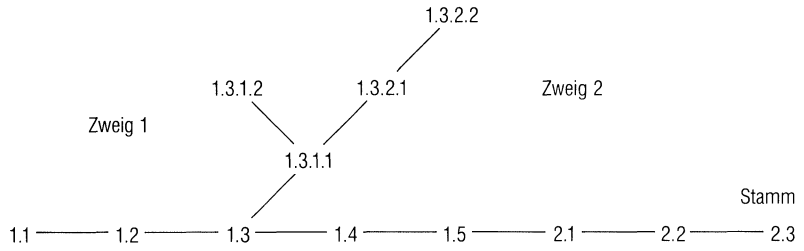


Bild 3-2 SID-Baum mit Zweigen

Im Knoten 1.3.1.1 beginnt ein neuer Zweig - der zweite, der von 1.3 abhängt. Er bekommt deshalb die Zweignummer 2. Die Folgenummer wird fortlaufend innerhalb des Zweigs gezählt. Nummern des ersten Zweigs haben immer die Form 1.3.1.n, die des zweiten Zweigs haben die Form 1.3.2.n.

Zusammenfassung:

Deltanummern bestehen immer aus genau

- zwei Komponenten, wenn sie Stammknoten sind,
- vier Komponenten, wenn sie Zweigknoten sind.

Aus der Deltanummer eines Zweigs können Sie

- erkennen, von welchem Stammknoten der Zweig abhängt,
- nicht erkennen, welche Abhängigkeiten innerhalb der Zweige bestehen.

3.1.4 Was Sie beim SCCS beachten sollten

- Sie sollten nicht zu viele Zweige einführen, um nicht den Überblick zu verlieren.
- Sie sollten SCCS-Dateien nur mit SCCS-Kommandos verwalten, verändern, lesen, vergleichen und holen.
- Falls mehrere Benutzer gleichzeitig auf eine SCCS-Datei zugreifen dürfen, wird ein Benutzer zum Verwalter ernannt. Der Verwalter kontrolliert die Zugriffsrechte. Nur er darf admin aufrufen.
- Wenn Sie mehrere Dateien, die voneinander abhängig sind, mit SCCS verwalten, sollten Sie die SID-Nummern konsistent halten, d.h. Sie
  - schreiben alle s-Dateien ins gleiche Dateiverzeichnis,
  - erstellen Deltas immer von allen Dateien gleichzeitig, selbst wenn Sie nicht alle Dateien verändert haben.
- Trotz SCCS müssen Sie die Dateien regelmäßig sichern.
- Versionen, die Sie an Kunden ausliefern wollen, holen Sie mit get ohne den Schalter -e oder -k. Wenn Sie im Text der Version dasIdentifikations-Schlüsselwort für die Versionsnummer (z.B. %I%, %W% oder %A%) angeben, ersetzt get das Schlüsselwort %I% durch die Versionsnummer. Der Kunde kann jederzeit abfragen, mit welcher Version er arbeitet (siehe what, Betriebssystem SINIX, Buch 1).

- Sie sollten neue Stamm-Versionen erstellen:
  - vor einer Freigabe oder Installation,
  - bei großen Veränderungen der Texte oder Programme (z.B. bei Änderung der internen Datenstruktur),
  - nicht allzu oft (z.B. nicht wegen Fehlerbehebungen oder kleinen Veränderungen).
- Eine zentrale Shell-Prozedur kann bei großen Dateisystemen die Verwaltung der SCCS-Dateien und der SCCS-Dateiverzeichnisse übernehmen.

Fehlerdiagnose

Die Fehlermeldungen der SCCS-Kommandos bestehen i.a. aus

- dem Wort "ERROR",
- dem Namen der betroffenen Datei,
- der Fehlerangabe,
- einer Fehlerkennziffer in runden Klammern.

Wenn ein schwerer Fehler auftritt, wird die Bearbeitung der aktuellen Datei abgebrochen und zur nächsten Datei, die angegeben ist, übergegangen.

Version 2.0

Die SCCS-Kommandos der Version 2.0 haben sich nicht geändert gegenüber den SCCS-Kommandos der Version 1.0C.

3.2 Die SCCS-Kommandos

Für die SCCS-Kommandos gelten dieselben Eingaberegeln wie für andere SINIX- oder CES-Kommandos.

Die folgende Übersicht teilt die Kommandos nach ihrer hauptsächlichen Funktion ein. Dabei kommen einige Kommandos mehrmals vor.

Informationen allgemein

bdiff	Große Dateien vergleichen
vc	Textdarstellung kontrollieren

Informationen über SCCS-Dateien

bdiff	Große Dateien vergleichen
get	Version aus einer SCCS-Datei holen
prs	Informationen über eine SCCS-Datei ausgeben
sccsdiff	Zwei Versionen einer SCCS-Datei vergleichen
val	SCCS-Dateien auf Konsistenz prüfen
what	Versionsnummern ausgeben

SCCS-Dateien verändern

Eigenschaften von SCCS-Dateien verändern

admin	SCCS-Dateien erstellen und verwalten

Versionen erstellen und verändern

admin	SCCS-Dateien erstellen und verwalten
cdc	Kommentar eines Deltas ändern
comb	SCCS-Deltas zusammenfassen
delta	Delta erstellen
get	Version aus einer SCCS-Datei holen
rmdel	Delta einer SCCS-Datei löschen
unget	get-Kommando rückgängig machen

SCCS-Dateien verwalten

Eigenschaften von SCCS-Dateien festlegen

admin	SCCS-Dateien erstellen und verwalten

SCCS-Dateien löschen und erstellen

admin	SCCS-Dateien erstellen und verwalten
rmdel	Delta einer SCCS-Datei löschen

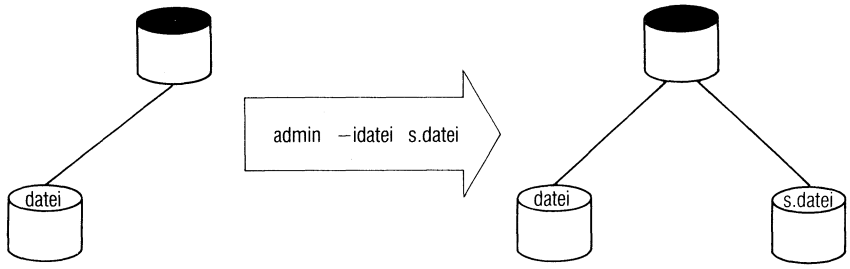
3.3 Vollständige Beschreibung der SCCS-Kommandos in alphabetischer Reihenfolge

Querverweise zu verwandten Kommandos finden Sie am Ende jeder Kommandobeschreibung, z.B.

>>>> ar verweist auf das genannte Kommando.

Die Beschreibung eines verwandten Kommandos kann auch in "Betriebssystem SINIX, Buch 1 (Kapitel 6)" stehen.

SCCS-Dateien erstellen und verwalten - create and administer SCCS files



admin

- erstellt neue SCCS-Dateien,
- initialisiert die Parameter einer neuen SCCS-Datei,
- ändert die Parameter einer SCCS-Datei, die bereits existiert,
- entdeckt und korrigiert nicht korrekte SCCS-Dateien,
- verwaltet SCCS-Dateien.

admin␣schalter...␣datei...

schalter

Die Reihenfolge der Schalter ist beliebig.

-n Sie wollen eine neue SCCS-Datei erstellen.

Wenn Sie nicht zusätzlich den Schalter -i angeben, können Sie mehrere neue SCCS-Dateien (ohne eigentlichen Text) einrichten.

-i[name] Sie wollen eine neue SCCS-Datei mit Text erstellen. Der Text für die erste Version steht in der Datei *name* (siehe auch Schalter -r).

Wenn *name* fehlt, erwartet admin, daß Sie den Text auf der Standard-Eingabe eingeben. Mit END beenden Sie die Texteingabe.

Der Schalter -i schaltet automatisch den Schalter -n ein.

Mit dem Schalter -i können Sie jeweils nur eine SCCS-Datei erstellen.

-rrel Sie sollten den Schalter -r nur zusammen mit dem Schalter -i oder -n verwenden.

rel ist die Releasenummer, die Sie dem ersten Delta geben wollen. Das Level des ersten Deltas ist immer 1. admin erstellt die Version rel.1.

Standard (keine Angabe):
rel = 1. Die erste Version hat die SID-Nummer 1.1.

-t[name] In der Datei *name* steht der "beschreibende Text" (siehe SCCS allgemein, "Format einer s-Datei") für die SCCS-Datei(en).

Wenn Sie eine SCCS-Datei neu einrichten und den Schalter -tangeben, dann darf *name* nicht fehlen.

Wenn die SCCS-Datei bereits existiert, dann wird der beschreibende Text, falls vorhanden,

bei Angabe -t	aus der SCCS-Datei entfernt,
bei Angabe -**t**name	in der SCCS-Datei ersetzt durch den Text in *name*.

-**f**param Der Schalter -f spezifiziert einen Parameter und eventuell einen Wert, den ein bestimmter Parameter der SCCS-Datei annehmen soll. Der Schalter -f kann die Wirkungsweise von einigen SCCS-Kommandos bis auf Widerruf (siehe Schalter -d) festlegen.

Sie können den Schalter -f mehrmals bei einem admin-Aufruf angeben.

param kann sein:

b Sie dürfen beim Kommando get den Schalter -b verwenden um Versionen mit einem Zweig (Branch) zu erstellen.

cmaxrel
maxrel, eine positive, ganze Zahl, muß kleiner als 10000 sein. *maxrel* bezeichnet die maximaleReleasenummer, die eine Version der SCCS-Datei annehmen darf.

Standard (keine Angabe):
maxrel = 9999.

fminrel
minrel, eine ganze Zahl, muß größer als 0 und kleiner als 9999 sein. *minrel* bezeichnet die minimaleReleasenummer, die eine Version der SCCS-Datei haben darf.

Standard (keine Angabe):
minrel = 1.

dSID Bei den folgenden get-Aufrufen soll *SID* die aktuelle Deltanummer sein. D.h. der Schalter -**r**SID ist für alle folgenden get-Aufrufe automatischeingeschaltet, bei denen Sie -r nicht explizit angeben.

get gibt eine Fehlermeldung aus, wenn *SID* nicht erlaubt oder möglich ist.

Mit $ admin -dd heben Sie die Deltanummer *SID* als aktuelle Deltanummer auf.

i Normalerweise gilt die Ausgabe

"No id keywords (cm7)"

bei get oder delta als Warnung.

Wenn Sie admin mit -fi aufrufen, dann gilt das Fehlen von Identifikations-Schlüsselwörtern ("id keywords") in der SCCS-Datei als schwerer Fehler.

j Für dieselbe Version einer SCCS-Datei dürfen gleichzeitig mehrere g-Dateien mit der Schutzbiteinstellung `-rw-r--r--` (Version 1.0B), bzw. `-rw-------` (Version 1.0C) existieren. Die g-Dateien müssen in verschiedenen Dateiverzeichnissen stehen. Veränderungen an ein und derselben Version können dann parallel erfolgen.

Ohne den Parameter j kann die Version einer SCCS-Datei,die ein Benutzer mit

```
$ get -e ...
```

geholt hat, kein zweites Mal mit $ get -e geholt werden. Erst wenn ein entsprechendes unget- oder delta-Kommando erfolgt ist, darf dieselbe Version wieder mit $ get -e geholt werden.

llist

list ist eine Liste von Releasenummern. Versionen mit den angegebenen Releasenummern dürfen nicht mehr verändert werden. Es ist nicht möglich, ein Delta, das auf diesen Versionen aufbaut, zu erstellen. Sie können auf keine der Versionen mit

```
$ get -e ...
```

zugreifen.

list hat das folgende Format:

rel[,*rel*...] oder
a

rel ist eine Releasenummer, die Sie sperren wollen.
Wenn "a" (Abkürzung für "alle") statt einer Releasenummer beim Schalter -l steht, dann können Sie keine Version der betreffenden SCCS-Datei mehr verändern.
Alle Releasenummern der SCCS-Datei sind zum Editieren gesperrt.

Beispiel

`$ admin -fla s.abc`	sperrt alle Versionen der Datei *s.abc*.
`$ admin -fl1,2,3 s.abc`	sperrt alle Versionen der Datei *s.abc*, die Releasenummer 1, 2 oder 3 haben.

n delta erstellt ein "Null"-Delta für die Releasenummern, die Sie übergehen, wenn Sie eine Version mit einer neuen Releasenummer einführen.

"Null"-Deltas dienen als Anknüpfpunkt für Zweignummern.

Wenn Sie z.B. nach Version 2.7 die Version 5.1 erstellen, dann übergehen Sie die Releasenummern 3 und 4. Fehlt der Schalter -fn, dann können Sie später keine Versionen 3.1.1.1 oder 4.1.1.1 erstellen.

qtext

Wenn Sie eine SCCS-Datei mit get (ohne Schalter -e oder -k) holen, wird das Identifikations-Schlüsselwort %Q% global im Text der SCCS-Datei durch *text* ersetzt.

text, eine ASCII-Zeichenreihe, darf kein Leer- oder Tabulatorzeichen enthalten. Zeichen, die eine besondere Bedeutung für die Shell haben, müssen Sie mit Gegenschrägstrich davor angeben. Andernfalls werden sie von der Shell wie üblich interpretiert (siehe Betriebssystem SINIX, Buch 1).

Beispiel

```
Kommando                    Was steht statt %Q% in der
                                  g-Datei?

$ admin -fqHallo! ...             Hallo!
$ admin -fq"Hallo!" ...           Hallo!
$ admin -fq"$name" ...          Gudrun (wenn name=Gudrun)
$ admin -fq\$\$\$ ...              $$$
$ admin -fq'$$$'                   $$$
```

mmod

Wenn Sie eine SCCS-Datei mit get (ohne Schalter -e oder -k) holen, wird das Identifikations-Schlüsselwort %M% global im Text der SCCS-Datei durch den Modulnamen *mod* ersetzt.

mod, eine ASCII-Zeichenreihe, darf kein Leer- oder Tabulatorzeichen enthalten. Zeichen, die eine besondere Bedeutung für die Shell haben, müssen Sie mit Gegenschrägstrich davor angeben. Andernfalls werden sie von der Shell wie üblich interpretiert (siehe Betriebssystem SINIX, Buch 1). Die Beispiele für den Schalter -fq gelten für den Schalter -fm entsprechend.

Standard (keine Angabe):
Der Modulname ist der Name der SCCS-Datei ohne "s." am Anfang.

ttyp

Wenn Sie eine SCCS-Datei mit get (ohne Schalter -e oder -k) holen, wird das Identifikations-Schlüsselwort %Y% global im Text der SCCS-Datei durch den Modultyp *typ* ersetzt.

typ, eine ASCII-Zeichenreihe, darf kein Leer- oder Tabulatorzeichen enthalten. Zeichen, die eine besondere Bedeutung für die Shell haben, müssen Sie mit Gegenschrägstrich davor angeben. Andernfalls werden sie von der Shell wie üblich interpretiert (siehe Betriebssystem SINIX, Buch 1). Die Beispiele für den Schalter -fq gelten für den Schalter -ft entsprechend.

v[pgm]

delta fragt nach MR-Nummern, wenn Sie ein neues Delta erstellen (MR ist die Abkürzung für "Modification Request"). Eine **MR-Nummer** steht z.B. stellvertretend für

- einen Grund, der angibt, warum Sie die SCCS-Datei einrichten,
- einen Änderungswunsch oder/und
- einen Änderungsgrund.

Sie können MR-Nummern angeben bei den Kommandos:

– $ admin -m ...	MR-Liste für die erste Version einer neuen SCCS-Datei festlegen
– $ cdc -m ...	MR-Liste einer Version ändern
– $ delta -m ...	MR-Liste für eine neue Version festlegen

pgm bezeichnet ein Programm, das prüft, ob MR-Nummern zulässig sind. *pgm* kann sowohl ein C-Programm als auch eine Shell-Prozedur sein (siehe Beispiel 3).

Wenn Sie eine neue SCCS-Datei einrichten und den Parameter v setzen, dann müssen Sie auch eine MR-Liste beim Schalter -m angeben. Die MR-Liste kann aber leer sein.

-**d**param

Sie wollen einen Parameter, den Sie vorher mit dem Schalter -**f**param eingeführt haben, wieder außer Kraft setzen. Mit dem Schalter -d können Sie deshalb nur SCCS-Dateien behandeln, die bereits existieren. Sie dürfen den Schalter -d bei einem admin-Aufruf mehrmals angeben.

param kann dieselben Parameter und Werte annehmen wie beim Schalter -f, z.B.

llist

list ist eine Liste von Releasenummern, die Sie nicht mehr sperren wollen für

```
$ get -e
```

list hat dasselbe Format wie beim Schalter -fl.

Sie müssen die Werte, die Sie den Parametern gegeben haben und die Sie außer Kraft setzen wollen, nicht immer angeben, z.B.

$ admin -dd	Falls Sie eine SID-Nummer für get-Aufrufe festgelegt haben, soll sie nicht mehr gelten.
$ admin -dl	Sie heben die Sperre für alle Releasenummern auf.

-**a**login *login* ist ein login-Name oder eine Gruppennummer (GID). admin trägt den login-Namen bzw. alle login-Namen, die die Gruppennummer *login* haben, in die Benutzerliste ein. Die Benutzer, die in dieser Liste stehen, dürfen

- die angegebenen SCCS-Dateien mit get -e holen,
- den Text in der g-Datei mit einem Editor verändern und
- neue Deltas erstellen,
- allgemein: die Kommandos cdc, comb, delta, get -e, unget, und rmdel verwenden.

Wenn die Benutzer-Liste leer ist, darf jeder Benutzer obige Kommandos aufrufen.

Bei einem admin-Kommando können Sie den Schalter -a beliebig oft angeben.

-**e**login *login* ist ein login-Name oder eine Gruppennummer. admin entfernt den login-Namen bzw. alle login-Namen, die die Gruppennummer *login* haben, aus der Benutzerliste. Die Benutzer sind nicht mehr berechtigt, die angegebenen SCCS-Dateien zum Editieren zu holen oder Deltas zu erstellen.

Bei einem admin-Kommando können Sie den Schalter -e beliebig oft angeben.

-**y**[kom] *kom* soll der Kommentar zum ersten Delta der SCCSDatei sein (ähnlich den Kommentaren beim Kommando delta). Sie können den Schalter nur zusammen mit den Schaltern -i und/oder -n angeben.

kom ist eine ASCII-Zeichenreihe, die zwischen Anführungsstrichen stehen muß, wenn sie Leer- oder Tabulatorzeichen enthält.

Fehlt der Schalter -y, dann bekommt das erste Delta z.B. den Kommentar

`date and time created 85/06/19 14:30:41 by gudrun`

wenn *gudrun* der login-Name der Benutzerin ist, die die SCCS-Datei anlegt.

-m[mrlist]

Sie wollen dem ersten Delta die MR-Nummern geben, die in *mrlist* stehen. Die MR-Nummern geben an, warum Sie die SCCS-Datei erstellt haben.

mrlist muß folgendes Format haben:

```
mrnummer1                                  wenn mrlist nur aus
                                           einer MR-Nummer besteht.

"mrnummer1␣mrnummer2␣...␣mrnummern"        wenn mrlist aus
          oder                             einer oder mehr
'mrnummer1␣mrnummer2␣...␣mrnummern'        MR-Nummern besteht.
```

mrnummer1,..., mrnummern sind die MR-Nummern. Zwischen den einzelnen MR-Nummern müssen ein oder mehrere Leer- oder Tabulatorzeichen stehen. MR-Nummern können beliebige druckbare ASCII-Zeichenreihen ohne Leer- oder Tabulatorzeichen sein.

Sie müssen den Schalter -fv immer zusammen mit -m einschalten. Wenn ein Prüfprogramm *pgm* für MR-Nummern vorliegt, überprüft admin die MR-Nummern der Liste, ob sie gültig sind. admin meldet einen Fehler, wenn -fv fehlt oder MR-Nummern nicht korrekt sind.

-h admin prüft die Struktur der SCCS-Datei und berechnet erneut die Prüfsumme (die logische Summe aller Zeichen der SCCS-Datei außer den Zeichen in der ersten Zeile). admin vergleicht die berechnete Summe mit der Prüfsumme, die in der ersten Zeile steht. Wenn die beiden Zahlen nicht übereinstimmen, gibt admin eine Fehlermeldung aus.

Der Schalter ist nur sinnvoll bei einer SCCS-Datei, die bereits existiert. admin ignoriert alle Schalter, die Sie zusätzlich angeben.

-z admin berechnet die Prüfsumme der SCCS-Datei und schreibt sie in die erste Zeile der SCCS-Datei (siehe Schalter -h). Sie erfahren nicht, ob die Prüfsumme vorher korrekt war.

Wenn Teile einer Datei (von Ihnen oder vom System) zerstört worden sind, können Sie mit dem Schalter -z die Datei wieder "SCCS-fähig" machen.

datei

Als *datei* können Sie den Namen einer s-Datei, eines Dateiverzeichnisses oder das Zeichen "-" angeben. admin behandelt den Namen eines Dateiverzeichnisses, wie wenn Sie die Namen aller Dateien des Verzeichnisses einzeln aufführen würden. admin ignoriert die Namen von Dateien, die keine s-Dateien sind oder für die Sie keine Leseberechtigung haben.

Wenn Sie statt *datei* "-" angeben, dann liest admin von der Standard-Eingabe. admin behandelt jede Eingabezeile, wie oben beschrieben, als Name einer Datei oder eines Dateiverzeichnisses.

Sie dürfen bei einigen Schaltern nur den Namen einer SCCS-Datei oder den Namen eines Dateiverzeichnisses mit nur einer SCCS-Datei angeben (siehe z.B. Schalter -i).

Wenn eine Datei noch nicht existiert, dann erstellt admin die Datei. Die Dateiparameter bekommen die Werte, die Sie bei den Schaltern angeben, oder Standard-Werte.

Wenn eine Datei bereits existiert, dann werden nur die Parameterwerte entsprechend geändert, die Sie angeben.

Hinweis

- SCCS-Dateien, die admin neu anlegt, haben die Schutzbiteinstellung `-r--r--r--` (Version 1.0B), bzw. `-r--------` (Version 1.0C) und den effektiven Benutzer als Eigentümer.
- Die comb-Prozedur interpretiert Zeichen, die für die Shell eine Bedeutung haben, wie üblich, wenn sie in den Zeichenreihen

  mod (Schalter -fm),
  text (Schalter -fq),
  typ (Schalter -ft)

  vorkommen. Nach den Kommandos

```
$ admin -ft'$name' s.datei
$ comb s.datei > shell
$ chmod +x shell
$ shell
$ get s.datei
```

  steht in der g-Datei *datei* statt %Y% z.B. "objektmodul", wenn name = objektmodul gilt.

Beispiele

1. SCCS-Datei erstellen:

   Die beiden Kommandos

   $

   $

   führen genau dasselbe aus. Der Text aus Datei *datei* bildet das erste Delta der SCCS-Datei *s.abc*. admin antwortet mit

   ```
   No id keywords (cm7)
   ```

   einer Warnung, die Sie ignorieren können.

   $

   Jetzt betrachtet admin das Fehlen von Identifikations-Schlüsselwörtern als Fehler und erstellt keine SCCS-Datei *s.abc*, wenn keine Identifikations-Schlüsselwörter im Text der ersten Version vorkommen.

   $

   Sie wollen statt Version 1.1 die neue SCCS-Datei mit Version 3.1 beginnen.

   Wenn admin die Datei *s.abc* korrekt erstellt hat, dann sollten Sie die Ursprungdatei *datei* löschen.

2. Kommentare zum ersten Delta angeben:

   Sie können Kommentare (-y) und/oder MR-Nummern (-m) zur Verfügung stellen, die angeben,

   - wann und warum eine SCCS-Datei erstellt wurde,
   - wer die SCCS-Datei erstellt hat.

   $

   Die MR-Nummer 17 steht stellvertretend für den Grund, warum Sie das erste Delta erstellt haben. Ein Kommentar fehlt. admin kommentiert das erste Delta in diesem Fall z.B. mit

   ```
   date and time created 85/06/19 14:30:41 by gudrun
   ```

In der Datei *beschr* steht ein Text, der z.B. den Inhalt und den Zweck der SCCS-Datei beschreibt. Dieser Text kann wesentlich ausführlicher sein als ein Kommentar, der nicht länger als eine Zeile sein sollte.

$

admin speichert den beschreibenden Text im entsprechenden Teil der SCCS-Datei ab. Mit

$

ersetzt admin den beschreibenden Text durch den Text, der in der Datei *neuebeschr* steht.

3. Ein Programm zur Verfügung stellen, das MR-Nummern prüft:

   Sie richten eine SCCS-Datei *s.qqq* ein mit

   $

   In der Datei *mrp* steht das übersetzte Prüfprogramm. Die MR-Nummern sollen nur Werte zwischen 0 und 10 annehmen. Ein einfaches C-Quellprogramm dafür lautet:

```
int atoi();
main(argc, argv)
int argc;
char *argv[];
{
 int m;
 while (argc > 1)
      {
        argc--;
        m = atoi(argv[argc]);
        if ((m < 0) || (m > 10))
            exit(1);          /* mindestens eine MR-Nummer nicht korrekt */
      }
 exit(0);                     /* MR-Nummern korrekt */
}
```

4. Nicht mehr korrekte oder unvollständige SCCS-Dateien erkennen und rekonstruieren

   Mit admin können Sie prüfen, ob Teile einer SCCS-Datei

   – zerstört (z.B. wegen Hardware- oder Systemfehlern) oder
   – nicht SCCS-gerecht behandelt (z.B. mit anderen als SCCS-Kommandos)

   wurden.

   Die Prüfsumme in der ersten Zeile einer SCCS-Datei dient dazu, zu erkennen, ob eine SCCS-Datei unvollständig oder nicht mehr korrekt ist. Die Prüfsumme gibt die (logische) Summe der Zeichen der SCCS-Datei an, ohne die Zeichen in der ersten Zeile mitzuzählen. Wenn Teile einer Datei zerstört sind, kann diese Zahl nicht mehr korrekt sein. Sie können die Datei nur dann mit anderen SCCS-Kommandos behandeln, wenn Sie die Prüfsumme vorher mit admin korrigiert haben. Kein SCCS-Kommando, außer admin mit Schalter -h oder -z, bearbeitet eine SCCS-Datei, die nicht mehr korrekt ist.

```
$
ERROR [s.datei]: corrupted file (co6)
$
```

   Die SCCS-Datei *s.datei* hat keine korrekte Prüfsumme mehr. Mit Hilfe eines Editors (bei kleineren Schäden) können Sie die Datei eventuell wieder rekonstruieren und reparieren. Das Kommando

```
$
```

   korrigiert die Prüfsumme. Schäden, die Sie nicht vollständig oder korrekt repariert haben, können Sie jetzt kaum mehr entdecken.

Dateien

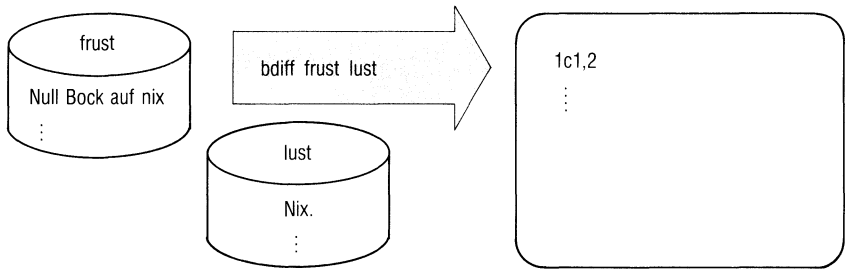
x-Datei	siehe Kapitel "SCCS allgemein"
z-Datei	siehe Kapitel "SCCS allgemein"

Siehe auch

L.E.Bonanni and C.A.Salemi,
Source Code Control System User's Guide

> > > > delta, ced, get, prs, what

Große Dateien vergleichen - big diff



bdiff vergleicht zwei Dateien genau wie diff. Der einzige Unterschied besteht darin, daß Sie mit bdiff Dateien bearbeiten können, die für diff zu groß sind.

bdiff gibt (genau wie diff) aus:

- die Zeilen, in denen sich die Dateien unterscheiden,
- ed-Kommandos, mit denen man aus *datei2 datei1* erzeugen kann.

bdiff␣datei1␣datei2[␣n][␣**-s**]

n

n ist eine ganze Zahl größer als 0. bdiff ignoriert am Anfang der Dateien die Zeilen, die sich nicht unterscheiden. Den Rest der Dateien teilt bdiff in Segmente von *n* Zeilen. bdiff ruft diff auf. diff vergleicht die entsprechenden Segmente der beiden Dateien. Sie müssen den Wert für *n* so wählen, daß diff keine Schwierigkeiten hat, Segmente mit *n* Zeilen zu vergleichen.

Standard (keine Angabe):
n = 3500

-s

bdiff gibt keine Fehlermeldungen aus. Allerdings zeigt diff Fehler an, die beim Vergleich der Segmente auftreten.

datei1 datei2

datei1 und *datei2* sind die Namen der Dateien, die Sie vergleichen wollen. Geben Sie statt eines Dateinamens das Zeichen "-" an, erwartet bdiff den Namen von der Standard-Eingabe.

Hinweis

- bdiff kann nur die Unterschiede innerhalb der entsprechenden Segmente in den Dateien feststellen, z.B. vergleicht bdiff nicht das 5. Segment in *datei1* mit dem 4. Segment in *datei2*. Deshalb gibt bdiff die Unterschiede oft umständlicher an als es notwendig wäre.
- Wenn Sie sowohl einen Wert für *n* als auch den Schalter -s angeben, dann ist die richtige Reihenfolge wichtig:

```
n muß vor -s stehen
```

Fehlerdiagnose

Siehe Schalter -s.

Beispiel

In der Datei *frust* steht folgender Text:

```
Null Bock auf nix.
Null Bock auf gar nix.
Null Bock auf überhaupt nix.
Holzbock.
```

In der Datei *lust* steht:

```
Nix.
Kein Bock auf nix.
Null Bock auf gar nix.
Null Bock auf überhaupt nix.
Holzbock.
```

```
$
```

führt zu der Ausgabe:

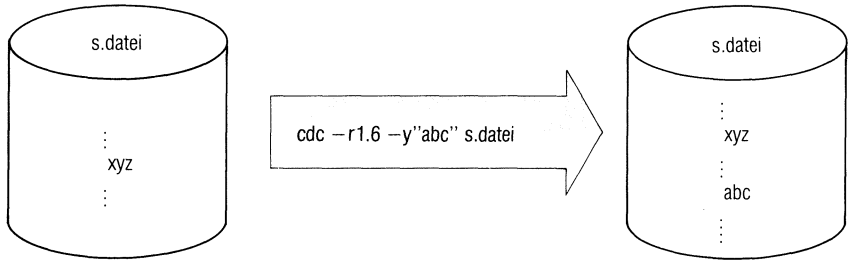
```
1c1,2
<   Null Bock auf nix.
---
>   Nix.
>   Kein Bock auf nix.
$
```

Dateien

/tmp/bd?????
Zwischendateien

> > > > diff, sccsdiff

Kommentar eines Deltas ändern - change the delta commentary of an SCCS file



Sie können zu jeder Version mit admin (für das erste Delta) oder delta (für die folgenden Deltas)

- einen Kommentar (Schalter -y) und/oder
- eine Liste mit MR-Nummern (Schalter -m)

übergeben. cdc ändert für die Version einer SCCS-Datei mit der Deltanummer *SID*

- den Kommentar und/oder
- die Liste mit den MR-Nummern.

cdc␣**-r**SID[␣schalter...]␣datei...

-**r**SID — cdc ändert den Kommentar und/oder die MR-Nummern der Version mit der Delta-Nummer *SID*. Die Version *SID* muß existieren.

schalter

-**m**[mrlist]

Wenn für eine SCCS-Datei der Parameter v gilt (siehe admin, Schalter -fv), dann können Sie zu den verschiedenen Versionen MR-Nummern angeben. Zu jeder Version gehört eine MR-Liste, in der diese MR-Nummern stehen.

mrlist ist eine Liste von MR-Nummern, die cdc in die MR-Liste der Version *SID* einfügen oder von der MR-Liste löschen soll. Wenn *mrlist* fehlt, hat der Schalter -m keine Wirkung.

mrlist muß folgendes Format haben:

```
mrnummer1                                  wenn Sie nur eine
                                           MR-Nummer angeben,

"mrnummer1␣mrnummer2␣...␣mrnummern"  wenn Sie eine oder
      oder                                 mehrere MR-Nummern
'mrnummer1␣mrnummer2␣...␣mrnummern'  angeben.
```

mrnummer1,..., mrnummern sind entweder MR-Nummern oder MR-Nummern, denen ein Ausrufzeichen "!" vorangestellt ist. Zwischen den einzelnen MR-Nummern muß mindestens ein Leer- oder Tabulatorzeichen stehen.

MR-Nummern können beliebige druckbare ASCII-Zeichenreihen sein, die keine Leer- oder Tabulatorzeichen enthalten dürfen.

cdc ergänzt die MR-Liste der Version *SID* um die MR-Nummern ohne "!" davor. cdc löscht MR-Nummern mit "!" davor, wenn sie in der aktuellen MR-Liste der Version stehen, und kommentiert in der SCCS-Datei, welche MR-Nummern gelöscht wurden.

Existiert zum Parameter v ein Wert *pgm*, also ein Programm oder eine Shell-Prozedur, dann prüft cdc, ob die MR-Nummern korrekt sind. Wenn eine MR-Nummer nicht korrekt ist, hat *pgm* einen return-Wert ungleich 0 und cdc ändert nichts.

Standard (kein Schalter -m, aber Parameter v):

Wenn Sie die Kommandos über eine Datensichtstation (als Standard-Eingabe) eingeben, erfolgt am Bildschirm die Frage

```
MRs?
```

und Sie können die MR-Nummern (eventuell mit "!" davor) interaktiv eingeben. Zwischen den einzelnen MR-Nummern müssen Sie mindestens ein Leer- oder Tabulatorzeichen schreiben. Mit der Taste ↵ beenden Sie die Eingabe der MR-Liste.
Die Frage "MRs?" kommt immer vor der Frage "comments?" (siehe unten).

Wenn die Standard-Eingabe keine Datensichtstation ist, unterläßt cdc die Frage.

-y[kom] Sie möchten den Kommentar der Version mit Nummer *SID* ändern. *kom*, ein beliebiger Text, soll der neue Kommentar sein. Wenn die Version bereits vorher einen Kommentar hatte, dann schreibt cdc in die SCCS-Datei, daß der frühere Kommentar nicht mehr gilt.

Wenn *kom* Leer- oder Tabulatorzeichen enthält, müssen Sie *kom* in Anführungsstriche (" oder ') setzen, z.B. -y"schon wieder ein Fehler!".

cdc ändert den Kommentar nicht, wenn *kom* fehlt.

Standard (kein Schalter -y):

Wenn Sie die Kommandos über eine Datensichtstation (als Standard-Eingabe) eingeben, erfolgt am Bildschirm die Frage

```
comments?
```

und Sie können den Kommentar interaktiv eingeben. Mit der Taste ↵ beenden Sie die Eingabe des Kommentars.
Die Frage "MRs?" kommt immer vor der Frage "comments?" (siehe oben).

Wenn die Standard-Eingabe keine Datensichtstation ist, unterläßt cdc die Frage.

datei

Als *datei* können Sie den Namen einer s-Datei, eines Dateiverzeichnisses oder das Zeichen "-" angeben. cdc behandelt den Namen eines Dateiverzeichnisses, wie wenn Sie die Namen aller Dateien des Verzeichnisses einzeln aufführen würden. cdc ignoriert die Namen von Dateien, die keine s-Dateien sind oder für die Sie keine Leseberechtigung haben.

Wenn Sie statt *datei* "-" angeben, dann liest cdc von der Standard-Eingabe. cdc behandelt jede Eingabezeile, wie oben beschrieben, als Name einer Datei oder eines Dateiverzeichnisses.

Sie können beliebig viele Namen angeben.

Hinweis

- Wenn Sie die Namen der Dateien, die cdc bearbeiten soll, über die Standard-Eingabe bereitstellen ("-"), müssen Sie die Schalter -m und -y beim cdc-Aufruf angeben. Sie können weder den Kommentar noch die MR-Nummern interaktiv eingeben.
- Welcher Benutzer berechtigt ist, einen Kommentar und/oder eine MR-Liste zu ändern, steht im Abschnitt "SCCS allgemein". Kurz gesagt, sind es die Benutzer,
  - die ein Delta erstellen dürfen,
  - die Eigentümer der betreffenden Datei und des Dateiverzeichnisses sind.

Beispiel

```
$ cdc -r1.6 -m"bl78-12345 !bl77-54321 bl79-00001" -y"Frohe Ostern!" s.datei
$
```

In der Version 1.6 der Datei *s.datei*

- erweitert cdc die MR-Liste um die MR-Nummern bl78-12345 und bl79-00001,
- entfernt cdc von der MR-Liste die MR-Nummer bl77-54321,
- ändert cdc den Kommentar ab in "Frohe Ostern!".

```
$ cdc -r1.6 s.datei
MRs? !bl77-54321 bl78-12345 bl79-00001
comments? Frohe Ostern!
$
```

hat genau dieselbe Wirkung wie das erste Kommando.

Dateien

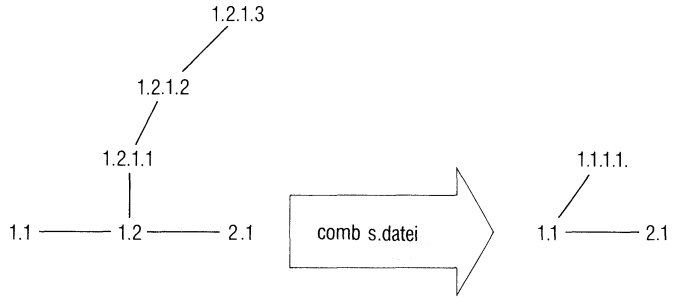
x-Datei	siehe Kapitel "SCCS allgemein"
z-Datei	siehe Kapitel "SCCS allgemein"

Siehe auch

L.E.Bonanni and C.A.Salemi,
Source Code Control System User's Guide

> > > > admin, delta, get, prs

SCCS-Deltas zusammenfassen - combine SCCS deltas



comb erstellt eine Shell-Prozedur und schreibt sie auf die Standard-Ausgabe. Mit der Shell-Prozedur können Sie den Aufbau einer SCCS-Datei so ändern, daß

- die geänderte s-Datei (hoffentlich) kleiner ist als die ursprüngliche s-Datei,
- Versionen, die Sie nicht mehr benötigen, zu Versionen zusammengefaßt werden, die für die Struktur der SCCS-Datei unbedingt nötig sind.

Die Shell-Prozedur ("comb-Prozedur")

- ändert den SID-Baum,
- vergibt laufende Nummern und SID-Nummern, die den neuen Baum widerspiegeln,
- speichert den ursprünglichen SID-Baum in der SCCS-Datei als "Beschreibenden Text" ab, so daß Sie kontrollieren können, wie sich der Aufbau verändert hat.

comb[␣schalter...]␣datei...

schalter

Die Reihenfolge der Schalter ist beliebig.

kein schalter

Die comb-Prozedur läßt nur Versionen übrig,

- die Blätter des SID-Baumes sind,
- die keine Blätter sind, aber unbedingt nötig sind, um den Baum zu erhalten.

Die comb-Prozedur entfernt Versionen, die beim Stamm oder bei den Zweigen irgendwo in der Mitte stehen, bzw. faßt die Versionen zu Deltas zusammen, die unbedingt nötig sind.

-pSID

Die comb-Prozedur entfernt alle Versionen, die älter sind als die Version mit der Nummer *SID*. Von den Versionen, die jünger oder gleich alt sind, läßt comb nur die Blätter und die unbedingt nötigen Versionen übrig.

-clist

In der Liste *list* führen Sie alle Versionen auf, die Sie behalten wollen. Die comb-Prozedur entfernt alle anderen Versionen. *list* hat folgendes Format

```
sid[,sid...]
```

sid kann sein

- eine vollständige SID-Nummer.
- der Teil einer SID-Nummer.
  comb bestimmt die vollständige SID-Nummer entsprechend der "Tabelle zur Bestimmung der SID-Nummer" beim Kommando get.
- ein Ausdruck der Form *sid1-sid2*.
  sid1 und *sid2* können vollständige oder Teil-SID-Nummern sein. *sid1-sid2* bezeichnet alle SID-Nummern zwischen *sid1* und *sid2*.

-o

In der Shell-Prozedur, die comb erstellt, wird normalerweise get -e mit dem Schalter -a aufgerufen. Der Aufbau der neuen SCCS-Datei erfolgt nach laufenden Nummern.

Wenn Sie bei comb den Schalter -o angeben, wird get -e in der Shell-Prozedur mit dem Schalter -r aufgerufen (statt -a). Der Aufbau erfolgt entsprechend der Nummer der Release, die gerade bearbeitet wird. Der SID-Baum kann eine stark veränderte Form bekommen. Beispiel 4 zeigt, welche Wirkung dieser Schalter hat.

-s

Die Shell-Prozedur, die comb erstellt, führt die Veränderung der SCCS-Datei nicht aus, sondern gibt folgende Informationen auf der Standard-Ausgabe aus:

- den Dateinamen
- den Umfang der Platzersparnis in Prozent
  100 * (urspr. Größe - neue Größe) / urspr. Größe
- die Größe der s-Datei (in Blöcken), wenn Sie die s-Datei mit der comb-Prozedur behandeln würden
- die Größe der ursprünglichen s-Datei (in Blöcken).

Sie sollten comb zuerst mit diesem Schalter (eventuell zusammen mit anderen) aufrufen um zu prüfen, ob die Shell-Prozedur überhaupt Platz sparen hilft.

datei

Als *datei* können Sie den Namen einer s-Datei, eines Dateiverzeichnisses oder das Zeichen "-" angeben. comb behandelt den Namen eines Dateiverzeichnisses, wie wenn Sie die Namen aller Dateien des Verzeichnisses einzeln aufführen würden. comb ignoriert die Namen von Dateien, die keine s-Dateien sind oder für die Sie keine Leseberechtigung haben.

Wenn Sie statt *datei* "-" angeben, dann liest comb von der Standard-Eingabe. comb behandelt jede Eingabezeile, wie oben beschrieben, als Name einer Datei oder eines Dateiverzeichnisses.

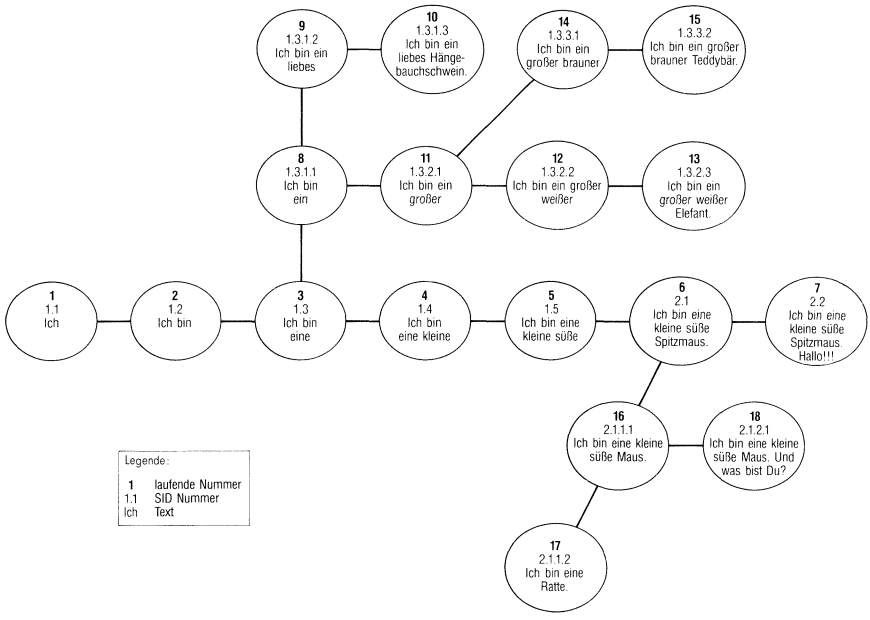
Sie können beliebig viele Namen angeben.

Hinweis

- Die comb-Prozedur kann den Aufbau des SID-Baums wesentlich ändern. Die geänderte SCCS-Datei benötigt manchmal sogar mehr Speicherplatz als die ursprüngliche SCCS-Datei.
- Sie sollten eine SCCS-Datei nur sehr selten mit der comb-Prozedur behandeln und immer zuerst comb mit dem Schalter -s aufrufen.

Beispiele

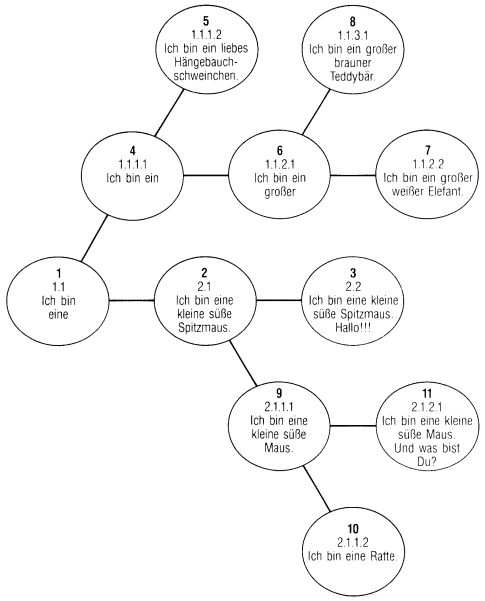
Der SID-Baum zur SCCS-Datei *s.jiri*:



In den Beispielen wird das comb-Kommando immer auf die Datei *s.jiri* mit diesem SID-Baum angewendet. Die Abbildung nach den Kommandos zeigt, wie sich der SID-Baum der SCCS-Datei *s.jiri* jeweils verändert hat.

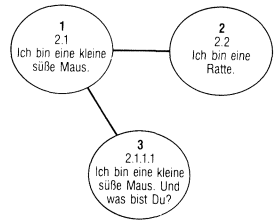
1. comb ohne Schalter

```
$ comb s.jiri > shell
$ chmod +x shell
$ shell
No id keywords (cm7)
$
```



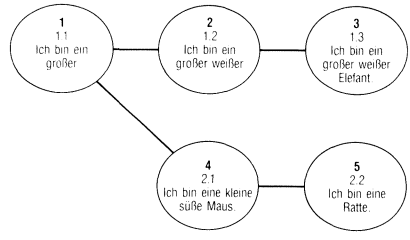
2. comb mit Schalter -p

```
$ comb -p2.1.1.1 s.jiri > shell
$ chmod +x shell
$ shell
No id keywords (cm7)
$
```



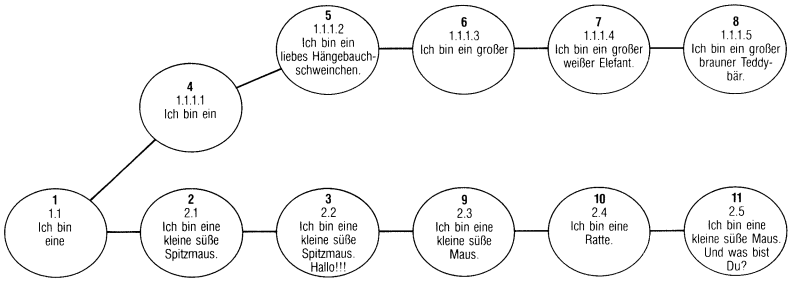
3. comb mit Schalter -c

```
$ comb -c1.3,2.1-1.3 2.3,2.1.1.1-2.1.1.2 s.jiri > shell
$ chmod +x shell
$ shell
No id keywords (cm7)
$
```



4. comb mit Schalter -o

```
$ comb -o s.jiri > shell
$ chmod +x shell
$ shell
No id keywords (cm7)
$
```



5. comb mit Schalter -s

```
$ comb -s s.jiri > shell
$ chmod +x shell
$ shell
No id keywords (cm7)
s.jiri   0%   3/3
$
```

Die Datei *s.jiri* wird nicht verändert.

Dateien

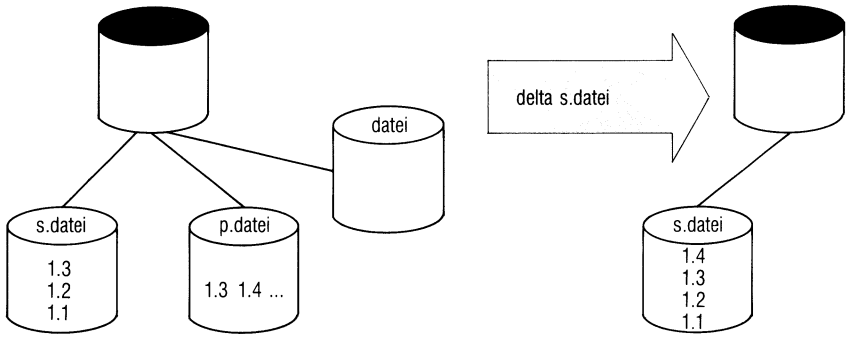
s.COMB?????	lokaler Name für die SCCS-Datei, solange die comb-Prozedur damit arbeitet.
comb?????	Zwischendateien

Siehe auch

L.E.Bonanni and C.A.Salemi,
Source Code Control System User's Guide

> > > > admin, delta, get, prs

Delta erstellen - make a delta (change) to an SCCS file



Mit delta erstellen Sie eine neue Version (ein neues Delta) für eine SCCS-Datei.

delta nimmt für die neue Version den Text, der in der g-Datei steht. Das Kommando

```
$ get -e ...
```

hat vorher die g-Datei errichtet. Die g-Datei besitzt denselben Namen wie die s-Datei, aber ohne "s." am Anfang. Sie müssen delta aufrufen, wenn Sie den Text der g-Datei mit einem Editor geändert haben und den geänderten Text als neue Version dem SCCS eingliedern wollen.

Ähnlich wie bei get -e prüft delta, ob ein Benutzer, der delta aufruft, auch dazu berechtigt ist.

Zusätzlich zur g-Datei erwartet delta, daß eine p-Datei existiert. delta prüft, ob in der p-Datei ein passender Eintrag mit Ihrem login-Namen steht.

Als nächstes kopiert sich delta die Vorgängerversion, auf der die neue Version aufbaut, aus der s-Datei. Die Vorgängerversion ist die Version, die get in die g-Datei geschrieben hat. Die Kopie steht in der d-Datei ("d" statt "s"). Danach prüft delta (mit bdiff), inwiefern Sie die g-Datei (mit einem Editor) gegenüber der d-Datei geändert haben.

delta stellt auf der Standard-Ausgabe eventuell Fragen, wenn bestimmte Schalter eingeschaltet oder bestimmte Parameter gesetzt sind (siehe -m, -y und admin).

delta schreibt auf die Standard-Ausgabe

- die Nummer der neuen Version,
- die Anzahl der Zeilen, die Sie gegenüber der Vorgängerversion eingefügt, gelöscht und nicht geändert haben.

Wenn delta die neue Version erstellt hat, wird der Eintrag aus der p-Datei gelöscht. Falls kein Eintrag mehr in der p-Datei ist, wird die p-Datei gelöscht. Andernfalls bleibt die p-Datei bestehen.

delta[␣schalter...]␣datei...

schalter

Die Reihenfolge der Schalter ist beliebig.

-rSID Mit *SID* legen Sie eindeutig fest, welches Delta Sie erstellen wollen. Sie können als *SID* angeben

– entweder die SID-Nummer der Version, die Sie neu erstellen wollen. Wenn Sie die Vorgängerversion mit

```
$ get -e
```

holen, steht *SID* nach "new delta" auf der Standard-Ausgabe.

– oder die SID-Nummer der Vorgängerversion. *SID* steht dann in der Zeile vor "new delta".

– oder die SID-(Teil-)Nummer, die Sie beim Schalter -r (-r*SID*) angegeben haben, als Sie die g-Datei erstellten.

Der Schalter -r ist nur nötig, wenn unter demselben login-Namen mehrere Versionen derselben SCCS-Datei zum Editieren (mit get -e) geholt wurden und entsprechende Einträge in der p-Datei stehen.

Beispiel

Die Version 2.1.1.1 der SCCS-Datei *s.abc* können Sie nach dem Kommando

```
$ get -e -r2.1 s.abc
2.1
new delta 2.1.1.1
3 lines
$
```

mit einem der folgenden delta-Aufrufe erstellen:

```
$ delta -r2.1 s.abc
      oder
$ delta -r2.1.1.1 s.abc
      oder
$ delta s.abc
```

delta meldet einen Fehler, wenn

- der Schalter -r unbedingt nötig ist und fehlt,
- die angegebene Versionsnummer SID nicht eindeutig oder unkorrekt ist.

Standard (keine Angabe):
delta erstellt die Version, die, entsprechend dem jüngsten Eintrag (mit Ihrem login-Namen) in der p-Datei, erstellt werden soll.

-s Auf der Standard-Ausgabe erscheint **nicht**

- die Nummer der neuen Version,
- die Anzahl der Zeilen, die Sie in der g-Datei eingefügt, gelöscht und geändert haben.

-n delta löscht nicht die g-Datei, die get erstellt hat. Normalerweise löscht delta die g-Datei im aktuellen Dateiverzeichnis.

-glist Versionen, deren Nummern in *list* angegeben sind, werden bei der neuen Version nicht berücksichtigt.

list hat folgendes Format

`sid[,sid...]`

sid kann sein

- eine vollständige SID-Nummer.
- der Teil einer SID-Nummer.
  delta bestimmt die vollständige SID-Nummer entsprechend der "Tabelle zur Bestimmung der SID-Nummer" beim Kommando get.
- ein Ausdruck der Form *sid1-sid2*.
  sid1 und *sid2* können vollständige oder Teil-SID-Nummern sein. *sid1-sid2* bezeichnet alle SID-Nummern zwischen *sid1* und *sid2*.

-m[mrlist]

Wenn für eine SCCS-Datei der Parameter v gilt (siehe admin mit Schalter -fv), dann müssen Sie zu einer neuen Version MR-Nummern angeben. Zu jeder Version gehört eine MR-Liste, in der diese MR-Nummern stehen. *mrlist* soll die MR-Liste der neuen Version SID sein.

mrlist muß folgendes Format haben:

```
mrnummer1                                  wenn Sie nur eine
                                           MR-Nummer angeben,

"mrnummer1␣mrnummer2␣...␣mrnummern"  wenn Sie eine oder
     oder                                  mehrere MR-Nummern
'mrnummer1␣mrnummer2␣...␣mrnummern'  angeben.
```

mrnummer1,..., mrnummern sind die MR-Nummern. Zwischen den einzelnen MR-Nummern müssen ein oder mehrere Leer- oder Tabulatorzeichen stehen.

Existiert zum Parameter v ein Wert *pgm*, also ein Programm oder eine Shell-Prozedur, prüft delta, ob die MR-Nummern korrekt sind. Wenn *pgm* einen return-Wert ungleich 0 hat, ist eine der MR-Nummern nicht korrekt und delta führt nichts aus (siehe admin, Beispiel 3).

Standard (kein Schalter -m, aber Parameter v):

Wenn Sie die Kommandos an einer Datensichtstation (als Standard-Eingabe) eingeben, erfolgt am Bildschirm die Frage

```
MRs?
```

und Sie können die MR-Nummern interaktiv eingeben. Ein oder mehrere Leer- oder Tabulatorzeichen trennen die einzelnen MR-Nummern. Mit der Taste ⏎ beenden Sie die Eingabe der MR-Liste.
Die Frage "MRs?" kommt immer vor der Frage "comments?" (siehe unten).

Wenn die Standard-Eingabe keine Datensichtstation ist, unterläßt delta die Frage.

-**y**[kom] *kom* soll der Kommentar zur neuen Version sein. Der Kommentar kann auch leer sein. Wenn *kom* Leer- oder Tabulatorzeichen enthält, müssen Sie *kom* in Anführungsstriche (" oder ') setzen.

Standard (kein Schalter -y):

Wenn Sie Ihre Kommandos an einer Datensichtstation (als Standard-Eingabe) eingeben, erfolgt am Bildschirm die Frage

```
comments?
```

und Sie können den Kommentar eingeben. Das Drücken der Taste ⏎ beendet die Eingabe des Kommentars.
Der Kommentar darf aus maximal 512 Zeichen bestehen. Wenn Sie innerhalb des Kommentars "Neue Zeile" schreiben wollen, müssen Sie das Fluchtsymbol "\" davor setzen.
Die Frage "MRs?" kommt immer vor der Frage "comments?" (siehe oben).

Wenn die Standard-Eingabe keine Datensichtstation ist, unterläßt delta die Frage.

-p

delta schreibt auf die Standard-Ausgabe die Unterschiede zwischen der d-Datei und der g-Datei. Die Unterschiede werden im selben Format wie bei diff, bdiff oder sccsdiff ausgegeben.

datei

Als *datei* können Sie den Namen einer s-Datei, eines Dateiverzeichnisses oder das Zeichen "-" angeben. delta behandelt den Namen eines Dateiverzeichnisses, wie wenn Sie die Namen aller Dateien des Verzeichnisses einzeln aufführen würden. delta ignoriert die Namen von Dateien, die keine s-Dateien sind oder für die Sie keine Leseberechtigung haben.

Wenn Sie statt *datei* "-" angeben, dann liest delta von der Standard-Eingabe. delta behandelt jede Eingabezeile, wie oben beschrieben, als Name einer Datei oder eines Dateiverzeichnisses.

Sie können beliebig viele Namen angeben.

Hinweis

- Wenn Sie die Namen der Dateien, die delta bearbeiten soll, über die Standard-Eingabe bereitstellen ("-"), müssen Sie die Schalter -y und -m beim delta-Aufruf angeben. Sie können weder den Kommentar noch die MR-Nummern über die Datensichtstation eingeben.
- Soll delta die Deltas zu mehreren Dateien erstellen, muß für alle Dateien entweder der Parameter v gelten oder nicht gelten. Bestimmend ist die Datei, die Sie als erstes nennen. delta ignoriert Dateien, die nicht mit der ersten Datei (bezüglich Parameter v) übereinstimmen.
- Welcher Benutzer berechtigt ist ein Delta zu erstellen, steht im Kapitel "SCCS allgemein".
- Wenn get große (p- und g-) Dateien einrichtet, sollten nicht zu viele get-Aufrufe nacheinander erfolgen. Stattdessen sollten Sie jedes get-Kommando mit einem entsprechenden delta-Kommando abschließen. Die Kommandofolge

  ```
  $ get -e    $ delta    $ get -e    $ delta    $ get -e    $ delta
  ```

  ist besser als die Folge

  ```
  $ get -e    $ get -e    $ get -e    $ delta    $ delta    $ delta
  ```

- Sie dürfen keine Zeile des Textes, den Sie in eine SCCS-Datei schreiben wollen, mit dem Zeichen CTRL A (ASCII-Zeichen mit Wert 1) beginnen. Das Zeichen hat eine besondere Bedeutung im SCCS.
- Der delta-Aufruf muß von dem Dateiverzeichnis aus erfolgen, in dem die g-Datei steht. Die SCCS-Datei muß dann mit dem entsprechenden Pfadnamen angegeben werden.

Beispiel

Sie haben in der Datei *englisch* folgenden Text stehen:

```
In Xanadu did Kubla Khan
a stately pleasure dome decree,
where Alph the sacred river ran,
through caverns measureless to man,
down to a sunless sea.
```

und bringen die Datei ins SCCS. Danach wollen Sie eine zusätzliche Anfangszeile einfügen.

```
$ get -e s.englisch
1.1
new delta 1.2
5 lines
$ ced englisch
.
.
.
$ delta -p -r1 2 -y"Zusätzliche Anfangszeile eingefügt" s.englisch
No id keywords (cm7)
1.2
0a1
> XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
1 inserted
0 deleted
5 unchanged
$
```

Dateien

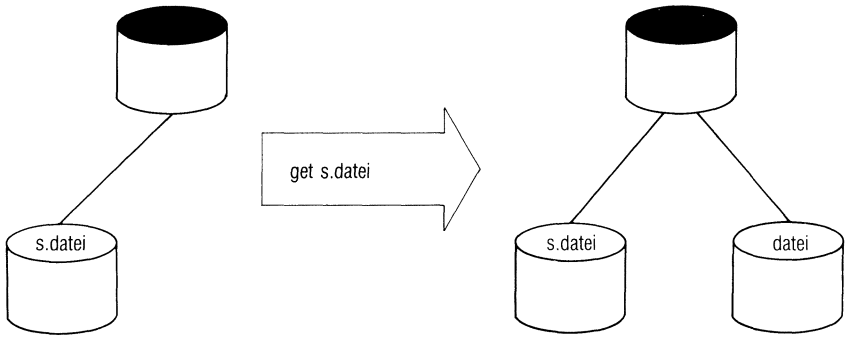
g-Datei	Die Datei existiert bereits vor dem Aufruf von delta (siehe get). Wenn delta korrekt beendet wird, wird die g-Datei im aktuellen Dateiverzeichnis gelöscht.
p-Datei	Die Datei existiert bereits vor dem Aufruf von delta (siehe get). Die Datei existiert eventuell noch, wenn delta beendet wird.
q-Datei	Die Datei wird während der Ausführung von delta erstellt und nach der Ausführung gelöscht.
x-Datei	Die Datei wird während der Ausführung von delta erstellt und nach der Ausführung umbenannt. Der neue Name ist der Name der s-Datei (siehe im Kapitel "SCCS allgemein").
z-Datei	Die Datei wird während der Ausführung von delta erstellt und gelöscht (siehe im Kapitel "SCCS allgemein").
d-Datei	Die Datei wird während der Ausführung von delta erstellt und nach der Ausführung gelöscht.
/usr/bin/bdiff	Programm, das die Unterschiede feststellt zwischen der g-Datei und der d-Datei.

Siehe auch

L.E.Bonanni and C.A.Salemi,
Source Code Control System User's Guide

> > > > admin, bdiff, get, prs, sccsdiff

Version aus einer SCCS-Datei holen - get a version of an SCCS file



get

- holt den eigentlichen Text einer bestimmten Version aus einer SCCS-Datei,
- erstellt eine **g-Datei** und
- schreibt den Text in die g-Datei.

Genau gesagt, holt get den eigentlichen Text der Anfangsversion der SCCS-Datei in die g-Datei. Darauf baut get solange Nachfolgerdeltas auf bis die gewünschte Version in der g-Datei steht.
Identifikations-Schlüsselwörter werden durch ihren Wert ersetzt (Ausnahme: Schalter -e und -k).

Die g-Datei hat denselben Namen wie die SCCS-Datei ohne "s." am Anfang.
Die g-Datei steht im aktuellen Dateiverzeichnis. Nur die Schalter -p und -g verhindern, daß die g-Datei erstellt wird.
Sie gehört dem realen Benutzer. Die Schutzbiteinstellung ist

-rw-r--r-- (Version 1.0B) -rw------- (Version 1.0C)	bei den Schaltern e und k. Sie können die g-Datei mit einem Editor verändern und dürfen neue Deltas erstellen.
-r--r--r-- (Version 1.0B) -r-------- (Version 1.0C)	sonst. Sie können die g-Datei lesen und z.B. übersetzen, dürfen Sie aber nicht mit einem Editor verändern.

get gibt für jede SCCS-Datei, deren Namen Sie angeben, auf der Standard-Ausgabe aus:

- den Namen der SCCS-Datei, wenn get mehr als eine Datei oder ein Dateiverzeichnis bearbeiten soll,
- die SID-Nummer der Version, die get in die g-Datei schreibt,
- "new delta" und die SID-Nummer der Version, die aus der Version in der g-Datei erstellt wird (nur beim Schalter -e),
- die Anzahl der Zeilen der Version, die get in die g-Datei schreibt,
- die Deltas, die get beim Schalter -i einfügt, nach dem Wort "Included:" (nur beim Schalter -i),
- die Deltas, die get beim Schalter -x ausschließt, nach dem Wort "Excluded:" (nur beim Schalter -x).

get[␣schalter...]␣datei...

schalter

Die Reihenfolge der Schalter ist beliebig.

-**r**SID Mit *SID* legen Sie die Nummer der Version fest, die get holt. *SID* kann eine vollständige oder auch unvollständigeSID-Nummer sein. Falls Sie aus der Version in der g-Datei eine neue Version bilden wollen (Schalter -e), legt *SID* auch die SID-Nummer der neuen Version fest. Die "Tabelle zur Bestimmung der SID-Nummer" zeigt für die wichtigsten Fälle, welche Version get jeweils holt und welche Version delta erstellt.

Standard (kein Schalter -r):

get holt die Version aus dem SCCS, die

- die höchste Releasenummer hat,
- im Stamm des SID-Baums steht und
- als letztes erstellt wurde.

Wenn Sie mit admin -fd eine SID-Nummer voreingestellt haben, dann holt get die Version mit dieser SID-Nummer.

-**c**cutoff *cutoff* ist eine Datumsangabe der Form

JJ[MM[TT[hh[mm[ss]]]]]

JJ, MM, TT, hh, mm, und *ss* sind zweistellige Zahlen, die das Jahr, den Monat, den Tag, die Stunde, die Minute, bzw. die Sekunde bezeichnen.

Wenn get die Anfangsversion aus dem SCCS holt und darauf Deltasaufbaut, berücksichtigt get keine Versionen und Änderungen (Deltas), die nach dem angegebenen Datum erstellt wurden. Angaben im Datum, die fehlen, bekommen den maximal möglichen Wert. Zwischen die zweistelligen Zahlen können Sie beliebig viele nichtnumerische Zeichen (außer Leer- oder Tabulatorzeichen) einfügen.

Beispiel

```
-c7502 bezeichnet das Datum -c750228235959,
-c77/02/02,09:22:25  bezeichnet das Datum -c770202092225.
```

-e

Sie wollen mit get eine Version der SCCS-Datei zum Editieren holen. Die Version soll die Grundlage einer neuen Version bilden. get schreibt die ursprüngliche Version in die g-Datei, die Sie ändern und mit delta wieder dem SCCS übergeben können.

Falls Sie die g-Datei verändern und dabei zerstören, können Sie die g-Datei mit get -k, statt get -e, rekonstruieren.

get erstellt die g-Datei nur, wenn alle Bedingungen erfüllt sind, die in der Parameterliste der betreffenden SCCS-Datei stehen. Z.B. muß der Benutzer, der eine Datei zum Editieren holt, in der Benutzerliste eingetragen sein (siehe admin -a). Außerdem benötigen Sie als realer Benutzer Schreiberlaubnis im aktuellen Dateiverzeichnis. Bevor get die SCCS-Datei holt, prüft get:

- ob der Benutzer, der get aufruft, in der Benutzerliste steht,
- ob $minrel \leq R \leq maxrel$,
  R Releasenummer der Version, die get holen soll,
- ob die Releasenummer R gesperrt ist,
- ob dieselbe Version der Datei bereits zum Editieren geholt wurde.

Zusätzlich schreibt get Informationen in die sog. **p-Datei**. Die Informationen sollen u.a. helfen, daß dieselbe Version nur dann noch einmal mit get -e geholt werden darf, wenn

- dem ersten get ein entsprechendes delta oder unget folgt,
- oder admin -fj vorher aufgerufen wurde.

Die p-Datei hat denselben Namen wie die s-Datei mit "p." statt "s." am Anfang. Die p-Datei steht im Dateiverzeichnis der s-Datei, für das der effektive Benutzer Schreiberlaubnis haben muß. Die p-Datei hat die Schutzbiteinstellung `-rw-r--r--`(Version 1.0B) bzw. `-rw-------` (Version 1.0C) und gehört dem effektiven Benutzer. Wenn die p-Datei schon existiert, erweitert get den Inhalt der p-Datei um eine Zeile. Sonst erstellt get die p-Datei. In einer Zeile der p-Datei steht:

- die SID-Nummer der Version, die get geholt hat;
- die SID-Nummer der Version, die erstellt werden soll;
- der login-Name des realen Benutzers;
- das Datum und die Uhrzeit, wann get ausgeführt wurde;
- die Deltaliste des Schalters -i, falls er eingeschaltet ist;
- die Deltaliste des Schalters -x, falls er eingeschaltet ist.

Die p-Datei darf beliebig viele Zeilen haben, aber in jeder Zeile muß die SID-Nummer der Version, die erstellt werden soll, verschieden sein.

Der Schalter -e schaltet automatisch den Schalter -k an.

-b Sie können diesen Schalter zusammen mit dem Schalter -e verwenden, wenn Sie einen neuen Zweig beginnen wollen. Die "Tabelle zur Bestimmung der SID-Nummer" beschreibt welche SID-Nummer das neue Delta haben wird. get ignoriert den Schalter, wenn

- der Parameter b nicht gesetzt ist (admin, Schalter -fb) oder
- das Delta, das get holt, kein Blatt am SID-Baum ist.

Sie können an einem Knoten des SID-Baums, der kein Blatt ist, auch ohne den Schalter -b einen Zweig beginnen.

-ilist Die Versionen aus *list* sollen beim Text, der in die g-Datei geschrieben wird, berücksichtigt werden.

list hat folgendes Format:

`sid[,sid...]`

sid kann sein

- eine vollständige SID-Nummer.
- der Teil einer SID-Nummer.
  get bestimmt die vollständige SID-Nummer entsprechend der "Tabelle zur Bestimmung der SID-Nummer", 4. Spalte.

– ein Ausdruck der Form *sid1-sid2*.
sid1 und *sid2* können vollständige oder Teil-SID-Nummern sein. *sid1-sid2* bezeichnet alle SID-Nummern zwischen *sid1* und *sid2*.

-**x**list Die Versionen aus *list* sollen beim Text, der in die g-Datei geschrieben wird, nicht berücksichtigt werden. *list* hat dasselbe Format wie beim Schalter -i.

Mit diesem Schalter können Sie Änderungen wieder rückgängig machen.

-k get ersetzt im Text der g-Datei die Identifikations-Schlüsselwörter nicht durch ihren Wert.

Der Schalter -e schaltet automatisch den Schalter -k ein. Die g-Datei enthält also beim Schalter -k denselben Text wie beim Schalter -e. Die p-Datei wird aber nicht verändert. Falls die g-Datei zerstört wird, während Sie sie verändern, können Sie die g-Datei mit -k wieder rekonstruieren und der Eintrag in der p-Datei ändert sich nicht.

-l[p] get schreibt eine Delta-Zusammenfassung in die sog. **l-Datei**, wenn der Schalter -l eingeschaltet ist.

get erstellt keine l-Datei sondern schreibt die Zusammenfassung auf die Standard-Ausgabe, wenn der Schalter -lp eingeschaltet ist.

Die l-Datei hat denselben Namen wie die s-Datei mit "l." statt "s." am Anfang. Die l-Datei enthält eine Tabelle der Deltas, die dazu beigetragen haben den Text in der g-Datei zu erstellen, d.h. welche Deltas eingefügt, ausgeschlossen und ignoriert wurden.
Die l-Datei steht im aktuellen Dateiverzeichnis, hat Schutzbiteinstellung `-r--r--r--` (Version 1.0B) bzw. `-r---------`(Version 1.0C) und gehört dem realen Benutzer. Nur der reale Benutzer muß im aktuellen Dateiverzeichnis Schreiberlaubnis haben.

Für jedes Delta stehen in der Tabelle mehrere Zeilen. Die erste Zeile hat folgendes Format:

–	Leerzeichen	wenn das betreffende Delta zum Text in der g-Datei beiträgt
	*	sonst
–	Leerzeichen	wenn das betreffende Delta ignoriert wird, egal ob es zum Text in der g-Datei beiträgt oder nicht
	*	wenn das betreffende Delta nicht zum Text beiträgt, aber auch nicht ignoriert wird
–	Grund	warum das betreffende Delta zum Text beiträgt oder warum nicht:
	"I"	Delta wird berücksichtigt (Schalter -i)
	"X"	Delta wird nicht berücksichtigt (Schalter -x)
	"C"	Delta wird abgeschnitten (Schalter -a, -b oder -c)
–	Leerzeichen	
–	SID-Nummer	
–	Tabulator	
–	Datum	an dem das Delta erstellt wurde Jahr/Monat/Tag Std.:Min.:Sek.
–	Leerzeichen	
–	Loginname	des Benutzers, der das Delta erstellt hat.

In den nächsten Zeilen folgen, um ein Tabulatorzeichen eingerückt, die MR-Liste und der Kommentar. Eine Leerzeile schließt jeden Eintrag ab.

-p get erstellt keine g-Datei sondern schreibt den Text auf die Standard-Ausgabe. Alles, was get normalerweise auf der Standard-Ausgabe ausgibt, schreibt get auf die Standard-Fehlerausgabe. Wenn zusätzlich der Schalter -s gesetzt ist, gibt get den Text auf der Standard-Ausgabe und nur noch die Fehlermeldungen auf der Standard-Fehlerausgabe aus.

-s get unterläßt alle Meldungen, die normalerweise auf die Standard-Ausgabe geschrieben werden. Fehlermeldungen werden auf der Standard-Fehlerausgabe ausgegeben.

-m Am Anfang jeder Textzeile, die get in die g-Datei schreibt, steht

- der Wert des Identifikations-Schlüsselworts %M% (Modulname), wenn der Schalter -n gesetzt ist;
- ein Tabulatorzeichen, wenn der Schalter -n gesetzt ist;
- die SID-Nummer des Deltas, das die betreffende Zeile in die SCCS-Datei eingefügt hat;
- ein Tabulatorzeichen.

Erst danach folgt der Zeilentext.

-n Am Anfang jeder Textzeile, die erstellt wird, steht

- der Wert des Identifikations-Schlüsselworts %M% (Modulname);
- ein Tabulatorzeichen;
- die SID-Nummer des Deltas, das die betreffende Zeile in die SCCS-Datei eingefügt hat, wenn zusätzlich der Schalter -m gesetzt ist;
- ein Tabulatorzeichen, wenn der Schalter -m gesetzt ist.

Erst danach folgt der Zeilentext.

-g get holt keinen Text aus der SCCS-Datei und erstellt keine g-Datei. Der Schalter dient z.B. dazu

- die SID-Nummer der jüngsten (Stamm-)Version zu erfahren (nur Schalter -g),
- eine l-Datei zu erstellen (Schalter -g und -l),
- eine p-Datei zu rekonstruieren, die zerstört wurde (Schalter -g und -e).

-t get holt das Top-Delta mit der angegebenen Releasenummer (z.B. -r1) oder Levelnummer (z.B. -r1.2).
Das Top-Delta ist die Version, die zeitlich als letztes erstellt wurde, unabhängig von der Lage im SID-Baum.

-**a**serno — *serno* ist die laufende Nummer der Version, die get holen soll. Die Shell-Prozedur, die comb erstellt, verwendet z.B. diesen Schalter.

Wenn Sie die Schalter -r und -a zusammen angeben, ignoriert get den Schalter -r. Wenn Sie die Schalter -e und -a zusammen verwenden, kann die Nummer der Version, die erstellt wird, anders als erwartet sein. Sie sollten diesen Schalter deshalb nur mit Vorsicht verwenden.

datei — Als *datei* können Sie den Namen einer s-Datei, eines Dateiverzeichnisses oder das Zeichen "-" angeben. get behandelt den Namen eines Dateiverzeichnisses, wie wenn Sie die Namen aller Dateien des Verzeichnisses einzeln aufführen würden. get ignoriert die Namen von Dateien, die keine s-Dateien sind oder für die Sie keine Leseberechtigung haben.

Wenn Sie statt *datei* "-" angeben, dann liest get von der Standard-Eingabe. get behandelt jede Eingabezeile, wie oben beschrieben, als Name einer Datei oder eines Dateiverzeichnisses.

Sie können beliebig viele Namen angeben.

Identifikations-Schlüsselwörter

Wenn get den Text einer Version aus der SCCS-Datei holt, ersetzt es die Identifikations-Schlüsselwörter im Text durch ihre Werte. Sie können so die Version, die geholt wurde, leichter identifizieren. Die Schlüsselwörter können an beliebiger Stelle im Text stehen.

Ausnahme
Die Schlüsselwörter werden **nicht** ersetzt, wenn der Schalter -k oderder Schalter -e einschaltet ist.

Tabelle der Identifikations-Schlüsselwörter

Identifikations-Schlüsselwort	Bedeutung des Schlüsselworts
%M%	Modulname: entweder der Wert *mod* bei admin -fm oder der Name der s-Datei ohne "s." am Anfang;
%I%	SID der geholten Version, gleichbedeutend mit %R%.%L%.%B%.%S%;
%R%	Releasenummer der Version;
%L%	Levelnummer der Version;
%B%	Zweignummer der Version (Branch);
%S%	Folgenummer der Version (Sequence);
%D%	aktuelles Datum (JJ/MM/TT);
%H%	aktuelles Datum (MM/TT/JJ);
%T%	aktuelle Uhrzeit (hh:mm:ss);
%E%	Datum, an dem das jüngste Delta erstellt wurde, das zu der Version beiträgt (d.h. letztes Änderungsdatum): JJ/MM/TT;
%G%	Datum, an dem das jüngste Delta erstellt wurde, das zu der Version beiträgt (d.h. letztes Änderungsdatum): MM/TT/JJ;
%U%	Zeit, zu der das jüngste Delta erstellt wurde, das zu der Version beiträgt: hh:mm:ss;
%Y%	Modultyp *typ* bei admin -ft;
%F%	Name der SCCS-Datei;
%P%	vollständiger Pfadname der SCCS-Datei;
%Q%	*text* bei admin -fq;
%C%	Aktuelle Zeilennummer (erleichtert die Fehlersuche);
%Z%	Zeichenreihe "@(#)", die das Kommando what erkennt;
%W%	Kurzform der what-Zeichenreihe %Z%%M%tab%I%, tab Tabulatorzeichen;
%A%	Kurzform der what-Zeichenreihe %Z%%Y%␣%M%␣%I%%Z%;

Tabelle zur Bestimmung der SID-Nummer

SID-Nummer, die Sie bei -r angeben	Schalter -b eingeschaltet?	andere Bedingungen	SID-Nummer der Version, die get holt	SID-Nummer der Version, die delta erstellt
keine Angabe	nein		mR.mL	mR.(mL+1)
keine Angabe	ja		mR.mL	mR.mL.(mB+1).1
R	nein	R > mR	mR.mL	R.1
R	nein	R = mR	mR.mL	mR.(mL+1)
R	ja	R > mR	mR.mL	mR.mL.(mB+1).1
R	ja	R = mR	mR.mL	mR.mL.(mB+1).1
R	ja, nein	R < mR, R existiert nicht	hR.mL	hR.mL.(mB+1).1
R	ja, nein	R < mR, R existiert	R.mL	R.mL.(mB+1).1
R.L	nein	R.L ohne Nachfolger im Stamm	R.L	R.(L+1)
R.L	ja	R.L ohne Nachfolger im Stamm	R.L	R.L.(mB+1).1
R.L	ja, nein	R.L hat Nachfolger im Stamm	R.L	R.L.(mB+1).1
R.L.B	nein	Zweig R.L.B existiert	R.L.B.mS	R.L.B.(mS+1)
R.L.B	ja	Zweig R.L.B existiert	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	nein	R.L.B.S Blatt des SID-Baums	R.L.B.S	R.L.B.(S+1)
R.L.B.S	ja	R.L.B.S Blatt des SID-Baums	R.L.B.S	R.L.(mB+1).1
R.L.B.S	ja, nein	R.L.B.S kein Blatt des SID-Baums	R.L.B.S	R.L.(mB+1).1

- "R", "L", "B", "S" sind die Release-, Level-, Branch-, bzw. Sequencenummer der SID-Nummer. "m" bedeutet "maximal existierende ...-Nummer".
  Z.B. bezeichnet "R.mL" die SID-Nummer mit der maximalen Levelnummer innerhalb der Release R. "R.L.(mB + 1).1" bezeichnet die SID-Nummer mit Sequencenummer 1 im neuen Zweig (mB + 1), mit Release R und mit Level L.

- Wenn Sie "R.L", "R.L.B.S" oder "R.L.B" angeben, muß ein entsprechender Knoten bzw. Zweig existieren.
- Wenn R nicht existiert, dann ist "hR" die höchste Releasenummer, die existiert, und die niedriger als R ist.
- Der Schalter -b hat nur dann eine Wirkung, wenn der Parameter b gesetzt ist (siehe admin, Schalter -fb).
- Wenn mit admin -fd eine SID-Nummer voreingestellt ist und Sie bei get den Schalter -r nicht angeben, dann gilt die Voreinstellung.

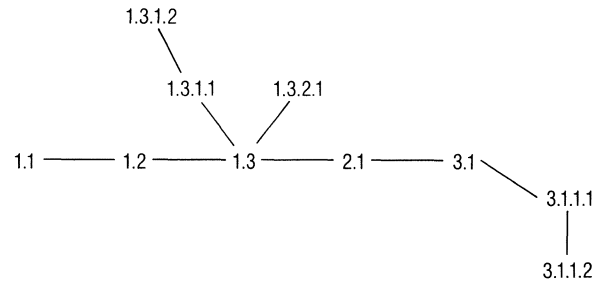
Hinweis

- Sie sollten die Schalter -i und -x mit größter Vorsicht anwenden, da es zu Überlagerungen bei verschiedenen Deltas kommen kann.
- Zu einer SCCS-Datei kann pro Dateiverzeichnis jeweils nur eine g-Datei existieren. Da die g-Datei immer im aktuellen Dateiverzeichnis erstellt wird, muß ein weiterer get-Aufruf (derselben SCCS-Datei) aus einem Dateiverzeichnis erfolgen, in dem noch keine entsprechende g-Datei existiert.

Beispiele

1. Welche Version holt get?

Die SCCS-Datei ***s.apollo*** soll folgenden SID-Baum erhalten:



Die SCCS-Kommandos dafür lauten:

```
$
No id keywords (cm7)
$
1.1
new delta 1.2
7 lines
$
comments?
No id keywords (cm7)
1.2
0 inserted
0 deleted
7 unchanged
$
1.2
new delta 1.3
...
$
...
$
```

```
1.3
new delta 2.1
...
$ delta s.apollo
...
$ get -e -r3 s.apollo
2.1
new delta 3.1
...
$ delta s.apollo
...
$ get -e -r1.3 s.apollo
1.3
new delta 1.3.1.1
...
$ delta s.apollo
...
$ get -e -r1.3 s.apollo
1.3
new delta 1.3.2.1
...
$ delta s.apollo
...
$ get -e -r1.3.1 s.apollo
1.3.1.1
new delta 1.3.1.2
...
$ delta s.apollo
...
$ get -e -r3.1 -b s.apollo
3.1
new delta 3.1.1.1
...
$ delta s.apollo
...
$ get -e -r3.1.1.1
3.1.1.1
new delta 3.1.1.2
...
$ delta s.apollo
...
```

2. get-Aufruf mit einem Dateiverzeichnis

   Im Dateiverzeichnis *XXX* stehen mehrere SCCS-Dateien.

```
$ get -e XXX
XXX/s.englisch:
2.2
new delta 2.3
6 lines

XXX/s.datei:
1.7
new delta 1.8
4 lines

XXX/s.grammatik:
2.4
new delta 2.5
240 lines
$ ls
XXX             datei           englisch        grammatik
$ cd XXX
$ ls
p.datei         p.grammatik     s.englisch
p.englisch      s.datei         s.grammatik
$ cd ..
$ delta XXX
comments? Nichts geändert!

XXX/s.englisch:
2.3
0 inserted
0 deleted
6 unchanged

XXX/s.datei:
...
```

3. g-Datei benennen

   Wenn Sie für die g-Datei einen Namen frei wählen wollen, müssen Sie die Schalter -p und -s verwenden.

```
$
...
```

4. Versionsnummer abfragen

   Die jüngste Version der SCCS-Datei *s.burma* enthält im Text das Identifikations-Schlüsselwort %A%. Sie holen die Version aus dem SCCS und fragen mit dem Kommando what die Versionsnummer ab.

```
$
1.3
7 lines
$
burma:
        burma 1.3@(#)
```

Dateien

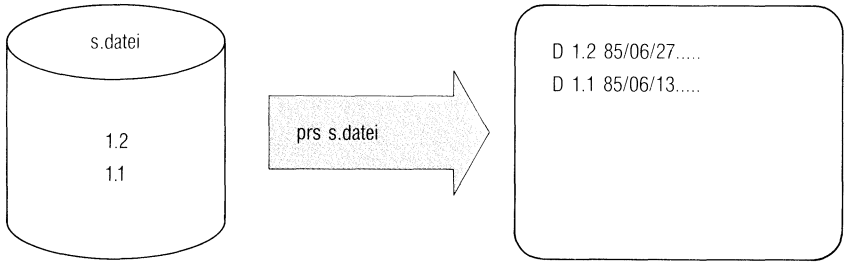
g-Datei	In der g-Datei steht der Text, den get aus dem SCCS holt.
l-Datei	Siehe Schalter -l[p]
p-Datei	Die Datei wird eventuell von get erstellt.
x-Datei	Siehe Kapitel "SCCS allgemein"
z-Datei	Siehe Kapitel "SCCS allgemein"

Siehe auch

L.E.Bonanni and C.A.Salemi,
Source Code Control System User's Guide

> > > > admin, delta, prs, unget, what

Informationen über eine SCCS-Datei ausgeben - print an SCCS file



prs gibt auf der Standard-Ausgabe Teile einer SCCS-Datei oder die gesamte SCCS-Datei aus. Sie können das Format festlegen, in dem die Ausgabe erfolgen soll.

prs[␣schalter...]␣datei...

schalter

Die Reihenfolge der Schalter ist beliebig.

kein Schalter

prs gibt für alle Versionen aus

- die Deltainformation,
- die Anzahl der Zeilen, die eingefügt, gelöscht und nicht geändert wurden,
- die MR-Liste,
- den Kommentar.

-d["datspez"]

Mit *datspez* bestimmen Sie das Ausgabeformat und spezifizieren,

- welchen Text und welche Daten
- prs in welcher Form ausgeben soll.

prs liefert folgende Informationen:

- den Text, den Sie in *datspez* angeben,
- die Werte der Daten-Schlüsselwörter (siehe unten), die in *datspez* vorkommen. prs holt den jeweiligen Wert aus der SCCS-Datei und gibt ihn statt des Schlüsselworts aus.
- wenn *datspez* fehlt:
  - die Deltainformation :Dt:,
  - die Anzahl der Zeilen, die eingefügt, gelöscht und nicht geändert wurden,
  - die MR-Liste,
  - den Kommentar.

datspez ist eine Zeichenfolge, die bestehen kann aus:

- SCCS-Daten-Schlüsselwörtern und/oder
- beliebigem sonstigen ASCII-Text.

Daten-Schlüsselwörter können beliebig oft und an beliebiger Stelle in *datspez* vorkommen.

Im sonstigen Text bedeutet "\t" Tabulatorzeichen und "\n" "Neue Zeile".

Wenn *datspez* nur aus einem Wort besteht, d.h. keine Leer- oder Tabulatorzeichen enthält, dann können Sie die Anführungsstriche davor und danach weglassen.

Beispiel

-ddatei:F: und -d"datei:F:" sind korrekte Schalter.

-**r**[SID] *SID* ist die Nummer des Deltas, über das Sie Information einholen wollen.

Standard (keine Angabe):
Die SID-Nummer der Version, die Sie als letztes erstellt haben.

-e prs soll Informationen ausgeben über

- die Version mit Nummer *SID* (siehe Schalter -r),
- alle Versionen, die Sie vor der Version mit Nummer *SID* erstellt haben.

-l prs soll Informationen ausgeben über

- die Version mit Nummer *SID* (siehe Schalter -r),
- alle Versionen, die Sie nach der Version mit Nummer *SID* erstellt haben.

-a Normalerweise gibt prs nur Information aus über Deltas, die existieren (Deltatyp D) und nicht über Deltas, die rmdel gelöscht hat (Deltatyp R). Mit diesem Schalter bekommen Sie auch Informationen über Deltas vom Typ R.

datei Als *datei* können Sie den Namen einer s-Datei, eines Dateiverzeichnisses oder das Zeichen "-" angeben. prs behandelt den Namen eines Dateiverzeichnisses, wie wenn Sie die Namen aller Dateien des Verzeichnisses einzeln aufführen würden. prs ignoriert die Namen von Dateien, die keine s-Dateien sind oder für die Sie keine Leseberechtigung haben.

Wenn Sie statt *datei* "-" angeben, dann liest prs von der Standard-Eingabe. prs behandelt jede Eingabezeile, wie oben beschrieben, als Name einer Datei oder eines Dateiverzeichnisses.

Sie können beliebig viele Namen angeben.

SCCS-Daten-Schlüsselwörter

SCCS-Daten-Schlüsselwörter geben an, welche Teile einer SCCS-Datei prs ausgeben soll. Sie können jeden Teil einer SCCS-Datei mit einem passenden Daten-Schlüsselwort ansprechen.

prs ersetzt ein Daten-Schlüsselwort, das in *datspez* vorkommt, durch seinen Wert. Das Format eines Wertes kann sein:

S prs gibt den Wert des Daten-Schlüsselworts aus,

M prs gibt den Wert des Daten-Schlüsselworts und das Zeichen "Neue Zeile" aus.

Tabelle der SCCS-Daten-Schlüsselwörter

Daten-Schlüssel-wort	Bedeutung des Daten-Schlüsselworts	Teil der s-Datei, in dem der Wert steht	Wert	Format
:Dt:	Deltainformation (für ein bestimmtes Delta)	Delta-Tabelle	*)	S
:DL:	Deltazeilenstatistik	Delta-Tabelle	:Li:/:Ld:/:Lu:	S
:Li:	Zeilen, die das Delta eingefügt hat	Delta-Tabelle	nnnnn	S
:Ld:	Zeilen, die das Delta gelöscht hat	Delta-Tabelle	nnnnn	S
:Lu:	Zeilen, die das Delta unverändert ließ	Delta-Tabelle	nnnnn	S
:DT:	Deltatyp	Delta-Tabelle	D oder R	S
:I:	SID-Nummer	Delta-Tabelle	:R:.:L:.:B:.:S:	S
:R:	Releasenummer	Delta-Tabelle	nnnn	S
:L:	Levelnummer	Delta-Tabelle	nnnn	S
:B:	Zweignummer (Branch)	Delta-Tabelle	nnnn	S
:S:	Folgenummer (Sequence)	Delta-Tabelle	nnnn	S
:D:	Datum, an dem das Delta erstellt wurde	Delta-Tabelle	:Dy:/:Dm:/:Dd:	S
:Dy:	Jahr, in dem das Delta erstellt wurde	Delta-Tabelle	nn	S
:Dm:	Monat, in dem das Delta erstellt wurde	Delta-Tabelle	nn	S
:Dd:	Tag, an dem das Delta erstellt wurde	Delta-Tabelle	nn	S
:T:	Uhrzeit, zu der das Delta erstellt wurde	Delta-Tabelle	:Th:::Tm:::Ts:	S
:Th:	Stunde, in der das Delta erstellt wurde	Delta-Tabelle	nn	S
:Tm:	Minute, in der das Delta erstellt wurde	Delta-Tabelle	nn	S
:Ts:	Sekunde, in der das Delta erstellt wurde	Delta-Tabelle	nn	S

Daten-Schlüs-sel-wort	Bedeutung des Daten-Schlüsselworts	Teil der s-Datei, in dem der Wert steht	Wert	For-mat
:P:	Benutzer, der das Delta erstellt hat	Delta-Tabelle	login-Name	S
:DS:	laufende Nummer des Deltas	Delta-Tabelle	nnnn	S
:DP:	laufende Nummer des Vorgängerdeltas	Delta-Tabelle	nnnn	S
:DI:	laufende Nummer der Deltas, die für das Delta nicht berücksichtigt, berücksichtigt, ignoriert wurden	Delta-Tabelle	:Dn:/:Dx:/:Dg:	S
:Dn:	laufende Nummer der Deltas, die berücksichtigt wurden	Delta-Tabelle	:DS:␣:DS:␣...	S
:Dx:	laufende Nummer der Deltas, die nicht berücksichtigt wurden	Delta-Tabelle	:DS:␣:DS:␣...	S
:Dg:	laufende Nummer der Deltas, die ignoriert wurden	Delta-Tabelle	:DS:␣:DS:␣...	S
:MR:	MR-Nummern des Deltas	Delta-Tabelle	Text	M
:C:	Kommentar	Delta-Tabelle	Text	M
:UN:	Benutzerliste	Benutzerliste	Text	M
:FL:	Parameterliste	Parameter	Text	M
:Y:	Modultyp (admin -ft)	Parameter	Text	S
:MF:	admin wurde mit Schalter -fv aufgerufen?	Parameter	yes oder no	S
:MP:	Name des Programms, das die MR-Nummern prüft (admin -fv)	Parameter	Text	S
:KF:	admin wurde mit Schalter -fi aufgerufen?	Parameter	yes oder no	S
:BF:	admin wurde mit Schalter -fb aufgerufen?	Parameter	yes oder no	S
:J:	admin wurde mit Schalter -fj aufgerufen?	Parameter	yes oder no	S
:LK:	Gesperrte Releasenummern (admin -fl)	Parameter	:R:␣... **)	S

Daten-Schlüs-sel-wort	Bedeutung des Daten-Schlüsselworts	Teil der s-Datei, in dem der Wert steht	Wert	For-mat
:Q:	Text bei admin -fq	Parameter	Text	S
:M:	Modulname (admin -fm)	Parameter	Text	S
:FB:	minimale Releasenummer (admin -ff)	Parameter	:R: **)	S
:CB:	maximale Releasenummer (admin -fc)	Parameter	:R: **)	S
:Ds:	Voreinstellung der Delta-nummer für get (admin -fd)	Parameter	:I: **)	S
:ND:	admin wurde mit Schalter -fn aufgerufen?	Parameter	yes oder no	S
:FD:	beschreibender Text	beschr. Text	Text	M
:BD:	eigentlicher Text	eigentl. Text	Text	M
:GB:	eigentlicher Text, mit get geholt	eigentl. Text	Text	M
:W:	Format einer "what"-Zei-chenreihe	Parameter/ Delta-Tabelle	:Z::M:\t:I:	S
:A:	Format einer "what"-Zei-chenreihe	Parameter/ Delta-Tabelle	***)	S
:Z:	Zeichenreihe, die das Kom-mando what erkennt	kein Teil der s-Datei	@(#)	S
:F:	Name der SCCS-Datei	kein Teil der s-Datei	Text	S
:PN:	vollständiger Pfadname der SCCS-Datei	kein Teil der s-Datei	Text	S

*) :DT:␣:I:␣:D:␣:T:␣:P:␣:DS:␣:DP

**) Diese Darstellung beschreibt nur das Format der Ausgabe. Sie können z.B. statt :FB: nicht :R: verwenden.

***) :Z::Y:␣:M:␣:I::Z:

Beispiele

1. prs mit Formatangabe

   Nach dem folgenden Kommando

```
$
```

   gibt prs aus:

```
Die Datei s.ente ist eine SCCS-Datei.
gudrun hat die neueste Version am 27.06.1985 erstellt.
Die Deltainformation lautet:
D 1.4 85/06/27 15:10:32 gudrun 7 6
```

2. prs ohne Formatangabe

   Um die Deltainformation aller Versionen zu bekommen, muß der Aufruf lauten:

```
$
s.ente:

D 1.4 85/06/27 15:10:32 gudrun 7 6        00000/00004/00008
MRs:
COMMENTS:
nein

D 1.3 85/06/27 10:55:38 gudrun 6 5        00001/00000/00011
MRs:
COMMENTS:
keine

D 1.2 85/06/27 10:49:45 gudrun 5 1        00000/00000/00011
MRs:
COMMENTS:
peter piper picked a peck of pickled peppers

R 1.3 85/06/14 15:22:32 gudrun 4 3        00000/00000/00011
MRs:
COMMENTS:
"%Z%%M%%I%"

R 1.2 85/06/14 15:04:20 gudrun 3 1        00000/00000/00011
MRs:
COMMENTS:
so ein unsinn
```

```
*** CHANGED *** 85/06/14 15:09:33 gudrun
kein unsinn

R 1.2 85/06/13 17:18:14 gudrun 2 1          00000/00000/00011
MRs:
COMMENTS:
nix

D 1.1 85/06/13 16:42:20 gudrun 1 0          00011/00000/00000
MRs:
COMMENTS:
date and time created 85/06/13 16:42:20 by gudrun

$
```

3. Benutzerliste ausdrucken

   Um die Benutzerliste auszudrucken, können Sie folgendes Kommando eingeben:

```
$ prs -d"Benutzerliste der SCCS-Datei :F:\n:UN:" s.abc
Benutzerliste der SCCS-Datei s.abc:
gudrun
nepomuk
stanislaus
godwina
albertine
$
```

Dateien

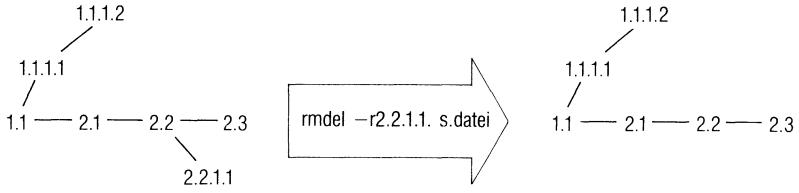
/tmp/pr?????
Zwischendateien

Siehe auch

L.E.Bonanni and C.A.Salemi,
Source Code Control System User's Guide

> > > > admin, delta, get, what

Delta einer SCCS-Datei löschen - remove a delta from an SCCS file



rmdel löscht eine Version aus einer SCCS-Datei. Die Version muß die jüngste Version sein, die innerhalb eines Zweigs (Branch) oder im Stamm des SID-Baumes erstellt wurde, d.h. sie muß ein Blatt des SID-Baumes sein. rmdel löscht keine Version, für die gerade ein neues Delta erarbeitet wird und für die ein Eintrag in der p-Datei existiert.

Der Typ des Deltas ändert sich von "D" (für "Delta") zu "R" (für "removed"). Der Typ eines Deltas steht in der Delta-Tabelle der SCCS-Datei.

rmdel␣-rSID␣datei...

SID — rmdel soll die Version mit der Versions- oder Delta-Nummer *SID* löschen. Sie müssen *SID* vollständig angeben, mit zwei oder vier Komponenten. Die Version darf nicht gesperrt sein (siehe admin, Schalter -fl). rmdel prüft auch, ob folgendes gilt (siehe admin, Schalter -fc, -ff):

minrel ≤ Releasenummer von *SID* ≤ *maxrel*

datei — Als *datei* können Sie den Namen einer s-Datei, eines Dateiverzeichnisses oder das Zeichen "-" angeben. rmdel behandelt den Namen eines Dateiverzeichnisses, wie wenn Sie die Namen aller Dateien des Verzeichnisses einzeln aufführen würden. rmdel ignoriert Namen von Dateien, die keine s-Dateien sind oder für die Sie keine Leseberechtigung haben.

Wenn Sie statt *datei* "-" angeben, dann liest rmdel von der Standard-Eingabe. rmdel behandelt jede Eingabezeile, wie oben beschrieben, als Name einer Datei oder eines Dateiverzeichnisses.

Sie können beliebig viele Namen angeben.

Hinweis

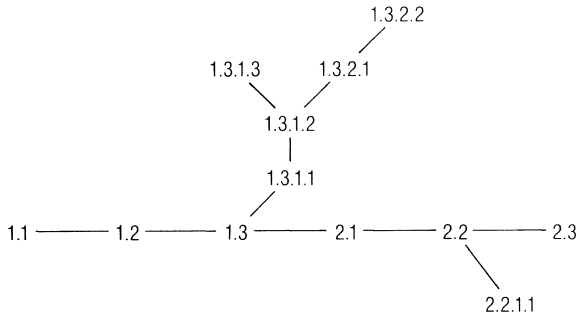
Welcher Benutzer berechtigt ist eine Version zu löschen, steht im Kapitel "SCCS allgemein". Kurz gesagt es muß der Benutzer sein,

- der das betreffende Delta erstellt hat oder allgemein Deltas für die SCCS-Datei erstellen darf (siehe Schalter -a und -e bei admin),
- der Eigentümer der betreffenden Datei und des Dateiverzeichnisses ist.

Der effektive Benutzer muß im Dateiverzeichnis der SCCS-Datei Schreiberlaubnis haben.

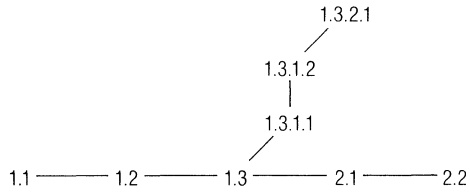
Beispiel

Sie haben eine SCCS-Datei mit folgendem SID-Baum erstellt:



Mit rmdel können Sie nur die Blätter des Baumes entfernen:

```
$ rmdel -r1.3.1.3 s.datei
$ rmdel -r1.3.2.2 s.datei
$ rmdel -r2.3 s.datei
$ rmdel -r2.2.1.1 s.datei
$
```



Als nächstes können Sie weitere Versionen entfernen:

```
$ rmdel -r1.3.2.1 s.datei
$ rmdel -r2.2 s.datei
$ rmdel -r2.1 s.datei
.
.
.
```

Dateien

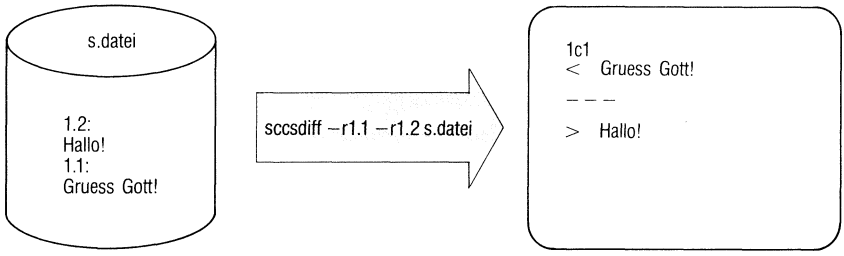
x-Datei	siehe Kapitel "SCCS allgemein"
z-Datei	siehe Kapitel "SCCS allgemein"

Siehe auch

L.E.Bonanni and C.A.Salemi,
Source Code Control System User's Guide

> > > > delta, get, prs, unget

Zwei Versionen einer SCCS-Datei vergleichen - compare two versions of an SCCS file



sccsdiff vergleicht zwei Versionen einer SCCS-Datei. Die Versionen werden an bdiff in der angegebenen Reihenfolge übergeben.

sccsdiff gibt (genau wie bdiff und diff) aus:

- die Zeilen, in denen sich die Versionen unterscheiden,
- ed-Kommandos, mit denen man aus Version *SID1* die Version *SID2* erzeugen kann.

sccsdiff␣**-r**SID1␣**-r**SID2[␣schalter...]␣datei...

SID1␣SID2 — *SID1* und *SID2* sind die SID-Nummern der Versionen (oder Deltas), die sccsdiff vergleichen soll. sccsdiff übergibt die Versionen in der angegebenen Reihenfolge an bdiff.

schalter

-p — Die Ausgabe von sccsdiff wird mit einer Pipe an pr weitergeleitet.

-sn
: *n* ist die Segmentgröße, die sccsdiff an bdiff übergibt. *n* muß eine ganze Zahl größer als 0 sein. bdiff ignoriert am Anfang der Dateien die Zeilen, die sich nicht unterscheiden. Den Rest der Dateien teilt bdiff in Segmente von *n* Zeilen. bdiff ruft diff auf. diff vergleicht die entsprechenden Segmente der beiden Dateien. Der Wert für *n* muß so gewählt sein, daß diff keine Schwierigkeiten hat, Segmente mit *n* Zeilen zu vergleichen.

  Standard (keine Angabe):
  n = 3500

datei
: *datei* ist der Name der s-Datei, deren Versionen sccsdiff vergleichen soll.

  Sie können beliebig viele Dateien angeben.

Hinweis

Da bdiff die Versionen in Segmente aufteilt, können nur die Unterschiede innerhalb der entsprechenden Segmente festgestellt werden. Nicht entsprechende Segmente werden nicht verglichen. Deshalb gibt bdiff die Unterschiede oft umständlicher aus als es notwendig wäre.

Fehlerdiagnose

Wenn sich die beiden Versionen nicht unterscheiden, erscheint die Meldung

`"datei: No differences"`

Beispiel

In der Datei *srinagar* steht folgender Text:

```
Null Bock auf nix.
Null Bock auf gar nix.
Null Bock auf überhaupt nix.
```

```
$
```

richtet die s-Datei *s.srinagar* ein. Mit

```
$
```

wird die SCCS-Datei geholt. Der Text in der editierten Datei *srinagar* wird geändert zu:

```
Nix.
Kein Bock auf nix.
Null Bock auf gar nix.
Null Bock auf überhaupt nix.
```

und mit

```
$
```

als Version 1.2 dem SCCS übergeben. sccsdiff stellt die Unterschiede der beiden Versionen fest:

```
$
1c1,2
<   Null Bock auf nix.
---
>   Nix.
>   Kein Bock auf nix.
$
```

Dateien

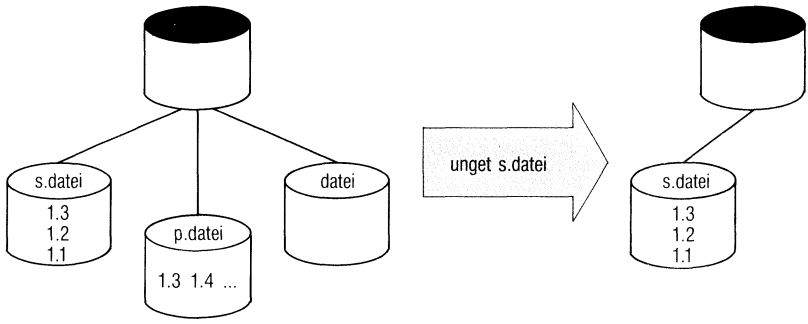
/tmp/get?????
Zwischendateien

Siehe auch

L.E.Bonanni and C.A.Salemi,
Source Code Control System User's Guide

> > > > bdiff, get, pr

Get-Kommando rückgängig machen - undo a previous get of an SCCS file



unget macht Ihr letztes Kommando

```
$ get -e ... datei
```

ungeschehen. Sie wollen nicht, wie vorher beabsichtigt, ein neues Delta erstellen.

unget[␣schalter...]␣datei...

schalter

Die Reihenfolge der Schalter ist beliebig.

-**r**SID Mit *SID* legen Sie eindeutig fest, welches Delta nicht mehr erstellt werden soll. Nach

```
$ get -e ...
```

erscheint diese SID-Nummer als "new delta" auf der Standard-Ausgabe. Der Schalter ist nur nötig, wenn unter demselben login-Namen mehrere Versionen derselben SCCS-Datei zum Editieren geholt sind und entsprechende Einträge in der p-Datei stehen.

unget meldet einen Fehler, wenn

- der Schalter -r unbedingt nötig ist und fehlt,
- die angegebene Versionsnummer *SID* nicht eindeutig oder unkorrekt ist.

Standard (keine Angabe):
SID-Nummer des Deltas, das unter Ihrem login-Namen als letztes erstellt werden sollte.

-s unget unterläßt es, die Nummer des Deltas auf der Standard-Ausgabe auszudrucken, das Sie nicht mehr erstellen wollen.

-n unget löscht nicht die g-Datei, die get erstellt hat. Normalerweise löscht unget die Dateien im aktuellen Dateiverzeichnis, die get angelegt hat.

datei Als *datei* können Sie den Namen einer s-Datei, eines Dateiverzeichnisses oder das Zeichen "-" angeben. unget behandelt den Namen eines Dateiverzeichnisses, wie wenn Sie die Namen aller Dateien des Verzeichnisses einzeln aufführen würden. unget ignoriert Namen von Dateien, die keine s-Dateien sind oder für die Sie keine Leseberechtigung haben.

Wenn Sie statt *datei* "-" angeben, dann liest unget von der Standard-Eingabe. unget behandelt jede Eingabezeile, wie oben beschrieben, als Name einer Datei oder eines Dateiverzeichnisses.

Sie können beliebig viele Namen angeben.

Hinweis

unget erwartet, daß im aktuellen Dateiverzeichnis die g-Datei zu der s-Datei steht, für die keine neue Version erstellt werden soll. Sie müssen daher unget aus demselben Dateiverzeichnis aufrufen, aus dem der entsprechende get-Aufruf erfolgte.

Beispiel

Sie haben die Version 1.3 der Datei *s.ente* aus dem SCCS geholt und wollen die Version 2.1 einführen. Aus einem anderen Dateiverzeichnis holen Sie die Version 1.3 um die Version 1.4 zu errichten.
Wenn Sie jetzt eine der beiden neuen Versionen nicht erstellen wollen, müssen Sie die entsprechende Versionsnummer angeben.

```
$
$
1.3
new delta 2.1
12 lines
$
ERROR [s.ente]: writable `ente' exists (ge4)
$
$
$
1.3
WARNING: being edited: `1.3 2.1 gudrun 85/06/27 14:08:26' (ge18)
new delta 1.4
12 lines
$
ERROR [../s.ente]: SID must be specified (un1)
$
1.4
$
```

Um auch noch den get-Aufruf im ersten Dateiverzeichnis rückgängig zu machen, brauchen Sie die SID-Nummer nicht anzugeben. unget soll die g-Datei nicht löschen.

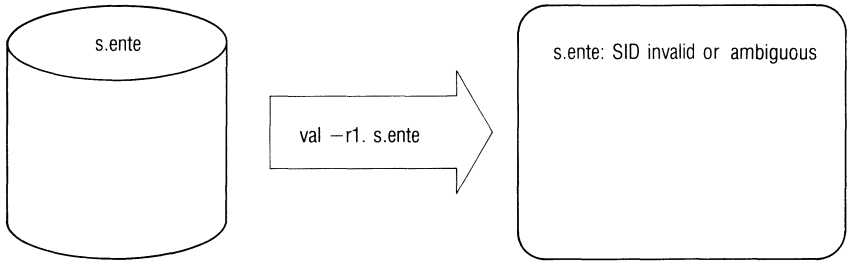
```
$
$
XXX                 ente                p.ente                s.ente
$
2.1
$
XXX                 ente                s.ente
$
```

Dateien

g-Datei	Siehe Schalter -n
p-Datei	unget löscht den entsprechenden Eintrag in der p-Datei, der nicht mehr gelten soll. Wenn die p-Datei danach leer ist, wird sie gelöscht.

> > > > delta, get, rmdel

SCCS-Dateien auf Konsistenz prüfen - validate SCCS file



val prüft die Konsistenz von SCCS-Dateien. Sie legen bestimmte Kriterien fest, die eine SCCS-Datei erfüllen soll. val betrachtet alle Kriterien, die nicht zutreffen, als Fehler. Die Fehler, die bei den angegebenen Dateien auftreten, schreibt val auf die Standard-Ausgabe.

Sie können die Dateinamen und die Schalter auch in mehreren Zeilen eingeben (siehe val -).

val[␣schalter...]␣datei...

schalter

Die Reihenfolge der Schalter ist beliebig.

-**r**SID *SID* sollte eine Deltanummer sein. val prüft, ob *SID*

- mehrdeutig oder
- ungültig ist.

Wenn beides nicht zutrifft, prüft val, ob *SID*

- existiert.

Beispiel

-r1	gibt nicht klar an, welche Version (z.B. 1.1, 1.2 oder 1.3) gemeint ist.
-r1.1.0.0	ist eine ungültige Deltanummer.

-s
val gibt auf der Standard-Ausgabe keine Fehlermeldungen aus.

-**m**mod
mod ist eine ASCII-Zeichenreihe. val vergleicht *mod* mit dem Modulnamen (Identifikations-Schlüsselwort %M%) in *datei* (siehe admin, Schalter -fm). *mod* darf keine Leer- oder Tabulatorzeichen enthalten.

-ytyp
typ ist eine ASCII-Zeichenreihe. val vergleicht *typ* mit dem Modultyp (Identifikations-Schlüsselwort %Y%) in *datei* (siehe admin, Schalter -ft). *typ* darf keine Leer- oder Tabulatorzeichen enthalten.

datei
datei ist der Name einer s-Datei.

val -

Wenn Sie val mit "-" aufrufen, dann erwartet val die Angabe der Schalter und Dateinamen auf der Standard-Eingabe. val betrachtet jede Zeile als Argumentenliste. Jede Zeile muß folgendes Format haben:

[␣schalter...]datei...

Wie üblich schicken Sie eine Zeile mit ↵ ab. Wenn Sie die Eingabe an val beenden wollen, drücken Sie END.

Ende-Status

val verwaltet ein 8-Bit-Wort, in dem alle Bits mit 0 vorbesetzt sind. Jedes Bit repräsentiert einen möglichen Fehler. val setzt ein bestimmtes Bit auf 1, wenn der entsprechende Fehler auftritt. Jedem Bit, das auf 1 gesetzt ist, entspricht ein Zahlenwert. Der Ende-Status ist die Summe der Zahlenwerte aller Bits, die auf 1 gesetzt sind. Der Ende-Status ist 0, wenn val bei keiner Datei, deren Namen Sie angeben, und in keiner Eingabezeile (bei val -) einen Fehler gefunden hat.

Bit	7	6	5	4	3	2	1	0
Zahlenwert	128	64	32	16	8	4	2	1

Bit	Fehler, bei dem das Bit auf 1 gesetzt wird	Fehlerausgabe von val
7	*datei* fehlt	: missing file argument
6	Schalterangabe unkorrekt	: unknown or duplicate keyletter argument
5	SCCS-Datei nicht SCCS-gemäß verändert	datei: corrupted SCCS file
4	*datei* kann nicht geöffnet werden oder ist keine SCCS-Datei	datei: can't open file or file not SCCS
3	SID ungültig oder mehrdeutig	datei: SID invalid or ambiguous
2	SID existiert nicht	datei: SID nonexistent
1	%Y% und *typ* verschieden	datei: %Y%, -y mismatch
0	%M% und *mod* verschieden	datei: %M%, -m mismatch

Hinweis

- val kann maximal 50 Dateinamen pro Kommandozeile bearbeiten.
- Die Syntax von *typ* und *mod* ist bei admin genauer beschrieben.

Beispiel

Die Datei *s.ente* soll den Modulnamen *entlein* und den Modultyp *text* bekommen.

```
$ admin -fmentlein -ftext s.ente
$ val -ytext -mentlein -r1.2 s.ente
```

val gibt keine Fehlermeldung aus. *s.ente* hat den Modulnamen *entlein*, ist vom Typ *text* und außerdem existiert die Version 1.2 von *s.ente*.

```
$ val -r"1.0a2" s.ente
    s.ente: SID invalid or ambiguous
$ echo $?
8
$
```

Sie können den Modultyp auch mit Anführungsstrichen eingeben. Versionsnummer 1. ist nicht korrekt. Der Ende-Status ist 8.

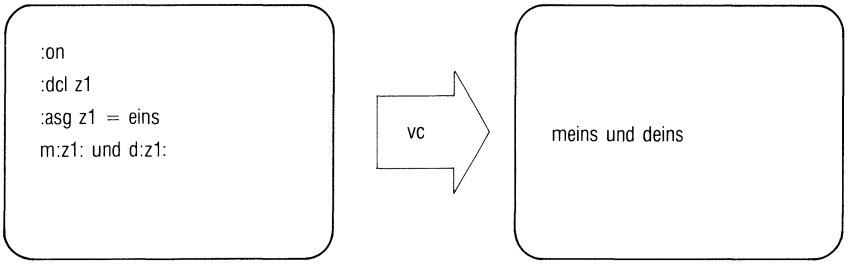
Sie können die obigen Kommandos auch so an val übergeben:

```
$ cat > val.cmd
-r"1.0a2"
s.ente
^D
$ val -y"text" -m"entlein" -f val.cmd
    s.ente: SID invalid or ambiguous
$ echo $?
8
$
```

Der Modultyp darf nicht in Anführungsstrichen stehen, da die Kommandos von val und nicht von der Shell gelesen werden.

> > > > admin, delta, get, prs

Textdarstellung kontrollieren - version control



vc kopiert ASCII-Text zeilenweise von der Standard-Eingabe auf die Standard-Ausgabe. Die Art und Weise der Kopie können Sie festlegen durch:

- Schalter
- Kontrollanweisungen, die Sie im Eingabetext einfügen und die zwischen Textzeilen stehen, die vc kopieren soll,
- Wertzuweisungen an Schlüsselwörter.

Jede Zeile der Eingabe enthält entweder einen Text, der kopiert werden soll, oder eine Kontrollanweisung.

Schlüsselwörter, die im Text und/oder den Kontrollanweisungen vorkommen, können durch die entsprechenden Werte ersetzt werden (siehe Schalter -a und Kontrollanweisungen).

vc ersetzt die Schlüsselwörter immer durch ihren Wert, wenn

– sie in einer Kontrollanweisung stehen und(!)

– ein Kontrollzeichen davor und danach geschrieben ist.

Kontrollzeichen ist standardmäßig ":". Mit dem Schalter -c können Sie ein anderes Zeichen als Kontrollzeichen wählen.

vc betrachtet eine Zeile (vorausgesetzt ":" ist Kontrollzeichen):

- als Kontrollanweisung, wenn die Zeile mit dem Zeichen ":" beginnt (siehe auch Schalter -t). vc kopiert die Zeile nicht.

- als Text, wenn sie mit "\:" beginnt. vc kopiert die Zeile ohne den ersten Gegenschrägstrich "\". Wenn Schlüsselwörter im Text ersetzt werden sollen (z.B. nach ":on"), dann erwartet vc noch einen zweiten Doppelpunkt in der Zeile. Ein Schlüsselwort am Zeilenanfang wird dann ersetzt.
- als Text, wenn sie mit "\" beginnt und das zweite Zeichen kein Doppelpunkt ist. vc kopiert die Zeile vollständig. Wenn Schlüsselwörter im Text ersetzt werden sollen (z.B. nach ":on"), betrachtet vc den Gegenschrägstrich als Fluchtsymbol. Der Gegenschrägstrich am Zeilenanfang wird nicht kopiert.
- als Text, wenn sie mit einem beliebigen ASCII-Zeichen, außer ":" oder "\", beginnt. vc kopiert die Zeile vollständig. Gegebenfalls werden Schlüsselwörter ersetzt.

vc[␣schalter...][␣schlüsselwort = wert...]

schalter

Die Reihenfolge der Schalter ist beliebig.

-a vc ersetzt auch alle Schlüsselwörter, die zwischen Kontrollzeichen in Textzeilen stehen. Der kopierte Text enthält dann statt der Schlüsselwörter die entsprechenden Werte. Ohne diesen Schalter ersetzt vc die Schlüsselwörter nur innerhalb von Kontrollanweisungen.

-t vc ignoriert alle Zeichen einer Zeile, vom Zeilenanfang an bis einschließlich dem ersten Tabulatorzeichen der Zeile, wenn der Rest der Zeile eine Kontrollanweisung darstellt, und führt die Kontrollanweisung aus. vc kopiert die gesamte Zeile, wenn der Rest der Zeile keine Kontrollanweisung ist.

-cchar Sie wollen *char* statt ":" als Kontrollzeichen verwenden.

-s vc unterdrückt Warnungen, die normalerweise auf der Standard-Fehlerausgabe ausgegeben werden. vc meldet weiterhin alle Fehler.

schlüsselwort

schlüsselwort ist eine alphanumerische ASCII-Zeichenreihe, die aus maximal 9 Zeichen bestehen darf. Das erste Zeichen muß ein Buchstabe sein.

wert

wert ist eine beliebige Zeichenreihe, die keine Leer- und Tabulatorzeichen enthalten darf und die ced erstellen kann. Numerische Werte werden als Zeichenfolgen von Ziffern ohne Vorzeichen betrachtet. Soll *wert* das Kontrollzeichen enthalten, so müssen Sie "\" als Fluchtsymbol verwenden.

Beispiel

Wenn vc das Schlüsselwort *olivia* durch den Wert "aa:aa" ersetzen soll, dann müssen Sie in der Kommandozeile angeben:

```
olivia=aa\:aa
```

Kontrollanweisungen

Kontrollanweisungen sind Eingabezeilen, die mit dem Kontrollzeichen ":" beginnen (siehe auch Schalter -t). Es gibt folgende Kontrollanweisungen:

:**dcl**␣schlüsselwort[,schlüsselwort...]

Alle Schlüsselwörter, die in der Eingabe vorkommen, müssen deklariert werden. Schlüsselwörter, denen Sie keinen Wert zuweisen, haben als Wert die leere Zeichenreihe.

:**asg**␣schlüsselwort = wert

vc weist dem Schlüsselwort (ähnlich wie in der Kommandozeile) einen Wert zu. Diese Kontrollanweisung überschreibt alle Wertzuweisungen an ein Schlüsselwort, die in der Kommandozeile oder in früheren Kontrollanweisungen vorkommen. Schlüsselwörter, denen kein Wert zugewiesen wird, haben als Wert die leere Zeichenreihe.

:**if**␣condition
.
.
.
:**end**

Wenn die Bedingung *condition* wahr ist, kopiert vc alle Zeilen, die zwischen der if-Anweisung und der end-Anweisung stehen. Wenn *condition* falsch ist, kopiert vc keine Zeile zwischen den beiden Anweisungen.
Kontrollanweisungen, die in den Zwischenzeilen vorkommen, werden ignoriert und nicht ausgeführt, wenn *condition* falsch ist.

condition hat die folgende Syntax (in yacc-artiger Notation):

```
condition  :  ["not"] A
           ;
A          :  B
           |  B "|" A                  /* log. oder */
           ;
B          :  ausdruck
           |  ausdruck "&" B           /* log. und */
           ;
ausdruck   :  "(" oder ")"
           |  wert op wert
           ;
op         :  "="                      /* gleich */
           |  "!="                     /* ungleich */
           |  "<"                      /* kleiner als */
           |  ">"                      /* größer als */
           ;
wert       :  ASCII-Zeichenreihe
           |  numerische Zeichenreihe
           ;
```

Links vom Doppelpunkt stehen die Namen der Elemente, aus denen sich *condition* zusammensetzen kann. Rechts steht welche Struktur die Elemente haben können. Zeichen, die zwischen Anführungsstrichen stehen, müssen Sie explizit (ohne die Anführungsstriche) angeben. "|" trennt die verschiedenen Alternativen für ein Element.
wert kann z.B. entweder eine ASCII-Zeichenreihe oder eine numerische Zeichenreihe sein.
condition kann entweder "not" *A* oder *A* sein.

Bedeutung der Operatoren:

"(" und ")" dienen dazu, um den Vorrang von Operatoren zu ändern oder logische Ausdrücke zusammenzufassen.

"not" darf nur direkt nach ":if" stehen (mit Leerzeichen dazwischen). "not" kehrt den Wahrheits-Wert der gesamten restlichen Bedingung um.

"<" und ">" vergleichen nur Integer-Werte ohne Vorzeichen. Alle anderen Operatoren behandeln die Werte als Zeichenreihen.

Die Operatoren "=", "!=", ">" und "<" haben gleichen Vorrang. "&" hat niedrigeren Vorrang und "|" hat den niedrigsten Vorrang.

Mindestens ein Leer- oder Tabulatorzeichen muß die Werte von den Operatoren oder Klammern trennen.

Beispiel

condition:	Wert von *condition*:
`not  777 < 766`	wahr
`not ( 777 > 766 )`	falsch
`012 > 12`	falsch
`012 != 12`	wahr
`( hans = uschi ) | ( a = a )`	wahr
`hans = uschi & a = a`	falsch

::text — *text* ist ein beliebiger ASCII-Text, den vc auf die Standard-Ausgabe kopiert. Die Doppelpunkte am Anfang der Zeile werden nicht kopiert. Schlüsselwörter, die im Text vorkommen und zwischen Doppelpunkten stehen, ersetzt vc durch ihre Werte. Dabei ist es egal, ob der Schalter -a eingeschaltet ist oder nicht.

:on — In den folgenden Textzeilen soll vc die Schlüsselwörter, die zwischen Kontrollzeichen stehen, durch ihre Werte ersetzen.

:off — In den folgenden Textzeilen soll vc die Schlüsselwörter nicht durch ihre Werte ersetzen.

:**ctl**␣char
char soll bis auf Widerruf das Kontrollzeichen sein.

:**msg**␣nachricht
vc soll *nachricht*, einen ASCII-Text, auf der Standard-Fehlerausgabe ausgeben.

:**err**␣nachricht
vc soll *nachricht*, einen ASCII-Text, auf der Standard-Fehlerausgabe ausgeben. Danach folgt eine Fehlermeldung von vc und die Angabe der Zeile, in der der Fehler aufgetreten ist. vc beendet seine Ausführung und hat den Ende-Status 1.

Ende-Status

0 — wenn vc normal beendet wird.
1 — wenn ein Fehler auftritt.

Beispiel

In der Datei *shona* steht folgender Eingabetext mit Kontrollanweisungen:

```
:on
:dcl z1
:dcl z2
:dcl z3
:asg z1=1
:asg z2=2
:asg z3=3

:if ( :z1: < :z2: )
Warum müssen Frauen schön sein und nicht klug?
Weil Männer besser sehen als denken können!
:end

:::z1:, :z2:, :z3:, fertig ist die Zauberei!
:ctl %
%asg z1=eins
%asg z2=ein
m:z1:, d:z1: und gar keins.
m%z1%, d%z1% und gar k%z1%.

%if ( 5 != 4 )
Du hast keine Chance.
Nutze sie!
%end

%if  hans = uschi  &  a = a
Am Anfang war das Wort.
%end

%if ( hans = uschi ) | ( a = a )
Aller Anfang ist schwer!
%end
%off

%err Schon wieder ein Fehler!
```

Nach dem Aufruf

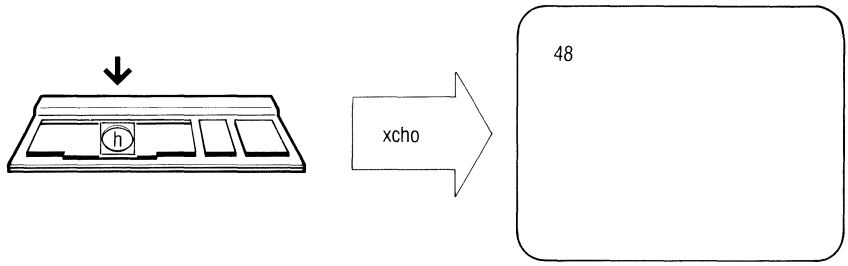
```
$ vc < snona
```

folgt die Ausgabe von vc

```
Warum müssen Frauen schön sein und nicht klug?
Weil Männer besser sehen als denken können!

1, 2, 3, fertig ist die Zauberei!
m:z1:, d:z1: und gar keins.
meins, deins und gar keins.

Du hast keine Chance.
Nutze sie!



Aller Anfang ist schwer!

ERROR: Schon wieder ein Fehler!
ERROR: err statement on line 29 (vc15)
$
```

vc bricht mit Ende-Status 1 ab.

Tastencode hexadezimal ausgeben - echo the keys typed in from the keyboard as hex digits



xcho gibt den ASCII-Code einer Taste der Bildschirmtastatur auf der Standard-Ausgabe hexadezimal aus, wenn Sie auf die betreffende Taste drücken.

xcho

Hinweis

- Die Standard-Eingabe muß immer die Tastatur einer Datensichtstation sein. xcho liest im cbreak-Modus ein.
- Sie können xcho mit CTRL D oder END beenden.

Beispiel

Nach dem Aufruf

```
$
```

folgt kein Bereitzeichen. Wenn Sie

an der Datensichtstation eingeben, erscheint bei jedem Tastendruck der Code der jeweiligen Taste am Bildschirm.

Bildschirm	Taste(n)
28	(
48	H
49	I
54	T
29	)
20	Leertaste
1a	CTRL Z
20	Leertaste
c	CTRL L
20	Leertaste
1b	↑
5b	
41	
20	Leertaste
1b	↓
5b	
42	
20	Leertaste
1b	←
5b	
44	
20	Leertaste
4	CTRL D
$	

A Anhang

Implementationsabhängiges Verhalten der verschiedenen Versionen

Version	2.0	1.0 B	1.0 C
Definitionsbereiche:			
int	+2147483647 bis −2147483648	+32767 bis −32768	+32767 bis −32768
short (int)	+32767 bis −32768	+32767 bis −32768	+32767 bis −32768
long (int)	+2147483647 bis −2147483648	+2147483647 bis −2147483648	+2147483647 bis −2147483648
unsigned int	0 bis 4294967295	0 bis 65535	0 bis 65535
unsigned short	0 bis 65535	0 bis 65535	0 bis 65535

Version	2.0	1.0 B	1.0 C
unsigned long	0 bis 4294967295	Typdeklaration unsigned long nicht möglich	
char	-128 bis +127	-128 bis +127	-128 bis +127
float	-3.4e38 bis +3.4e38	-1e-38 bis +1e38	-1e-38 bis +1e38
double	-1.8e308 bis +1.8e308	-1e-38 bis +1e38	-1e-38 bis +1e38

Sonstiges:

Version	2.0	1.0 B	1.0 C
Registeranzahl für Speicherklasse register	5	2	2
Darstellung negativer Zahlen	2er Komplement	2er Komplement	2er Komplement
Vorzeichen des Restes bei integer Division	positiv	positiv	positiv
Typdeklaration short int möglich	ja	nein (nur short möglich)	
Konvertierungsverhalten int -> short int	Es werden die niederwertigen 16 Bit berücksichtigt.	Kein Unterschied, da int und short intern gleich dargestellt werden.	
Konvertierungsverhalten negative float -> integer	Programmabbruch	Liegt der ganzzahlige Teil im Definitionsbereich von int, wird konvertiert. Sonst Programmabbruch.	
Zeiger als int	unproblematisch	unproblematisch, solange Zeiger innerhalb des int Definitionsbereichs liegt, sonst Überlauf (negativ).	

Version	2.0	1.0 B	1.0 C
Maximale Größe eines Vektors	[2147483646]	[32767]	[32767]
Maximale Anzahl von case-Anweisungen innerhalb einer switch-Anweisung	200	230	230
maximale Zeiger-Verschachtelungstiefe	>125	6	6
NULL in stdio.h enthalten	ja	ja	ja
Suchmethode für Include-Dateien	Wird der Dateiname von "<" und ">" eingeschlossen, erfolgt die Suche nur im Dateiverzeichnis /usr/include. Wird der Name zwischen Anführungsstrichen (") eingeschlossen, wird zuerst im Dateiverzeichnis des Programms, das übersetzt werden soll, gesucht. Nach erfolgloser Suche wird /usr/include durchsucht.		

Fachwörter deutsch - englisch

Abbildung	map
abfangen (Signal)	catch
Ablauf verfolgen	trace
ablauffähiges Programm	executable program
absolutes Symbol	absolute symbol
Abstand	offset
Adresse	address
Adreßraum	address space
ändern	change
Änderungsgrund	MR, modification request
Änderungswunsch	MR, modification request
Aktion	action
aktuell	current
aktuelles Dateiverzeichnis	working directory, local directory
aktuelles Umfeld	current environment
akzeptieren	accept
Alarmuhr	alarm clock
Anfrage	request
Anzeiger	flag
Argument	argument
Assembler	assembler
Assembler-Quellcode	assembly source program
assembliertes Programm	object code
assoziativ	associative
aufrufen	call
ausbessern	patch
Ausdruck	expression
ausführen, durchsuchen	execute
Ausführberechtigung	execute permission
Ausführung	execution
Ausgabe	output
aushängen (Dateisystem)	umount
austauschen	swap
Automat	automaton
automatisch	automatic
Basisadresse	base address
Baum	tree
Befehl	instruction
Benutzernummer	user ID (UID)
Benutzerzeit	user time
Bereinigung	cleanup
Bereitzeichen	prompt
beschreibender Text	descriptive text
Bibliothek	library, archive

Bildschirm	screen
Bildschirm, Datensichtstation	terminal
Bildschirmgruppe	tty group
Binärdatei	binary file
Binder	loader
Blatt	leaf
blockorientiertes Gerät	block special device
Branch	branch
break	break
break-Anweisung	break
Bruchteil	fractional part
bss-Segment	bss
Byte = 8 Bit	byte
cast	cast
cast-Anweisung	cast
common-Symbol	common symbol
Datei	file
Datei mit FILE-Struktur	stream
Dateiadresse	file address
Dateiende	end-of-file
Dateikennzahl	file descriptor
Dateiname	file name
Dateistatus	file status
Dateisystem	file system
Dateiverzeichnis	directory
Dateizeiger	file pointer
Datensegment	data area, data segment
Datensegment, nicht initialisierte Daten	bss
Datensymbol	data symbol
Delta	Delta, Version
Deltanummer	SID
deterministisch	deterministic
dezimal (10)	decimal
Diskette	floppy disk
doppelte Gleitpunktzahl	double floating point
dual (2)	dual
Dummy	dummy
dynamisch	dynamic
effektive Benutzernummer	effective UID
effektive Gruppennummer	effective GID
Eigentümer	owner
einfügen	insert
Eingabe	input
Eingabestrom	input stream
einhängen (Dateisystem)	mount
einstelliger Operator	monadic operator
Endestatus	exit status
endlicher Automat	finite automaton

Ergebnis	result
externe Referenz	external references
externes Symbol	external symbol
Fehler	error, bug
Fehlerbehandlung	error handling
Fehlercode	error code
Fehlermeldung	error diagnostic
Fehlersuche	debugging
Feld, Vektor	array
Festplatte	disk
FILE-Struktur	FILE structure
Flucht	escape
Folgenummer	sequence
Format	format
formatierte Ein/Ausgabe	formated I/O
freigeben	free
Gegenschrägstrich	backslash
gepufferte Ein/Ausgabe	buffered I/O
Gerät, externer Datenträger	device
Gerätedatei	special file
Gleitkommazahl	floating point number
Gleitpunktzahl	floating point number
global	global
Grammatik	grammar
Greenwich Meantime	GMT
Größe	size
Gruppe	group
Gruppenname	groupname
Gruppennummer	group ID (GID)
Haltepunkt	breakpoint
hexadezimal (16)	hexadecimal
Home-Dateiverzeichnis	home directory
Identifikations-Schlüsselwort	id keyword, identification keyword
include-Datei	include file
Indexeintrag	inode
Indexnummer	inumber
Inhaltsverzeichnis	table of contents
initialisieren	initialize
Inkrement	increment
Keller	stack (LIFO)
Knoten	node
Konsole	console
Kontextabhängigkeit	context sensitivity
kontextfrei	context-free
Kontrollanweisung	control statement
Kontrollzeichen	control character
Korrekturtaste	back space key
laufende Nummer	sequence

Laufzeit	elapsed time
Laufzeit-Startfunktion	runtime startoff
Laufzeitinkonsistenz	phase error
Leerzeichen	blank
Leseerlaubnis	read permission
lesen	read, shift
Level	level
Levelnummer	level
lexikalische Analyse	lexical analysis
löschen	delete
login-Name	login name
LR(1)-Parsing-Algorithmus	LR(1) parsing algorithm
magic number	magic number
major Nummer (Gerätetyp)	major number
Makro	macro
Maske	mask
mehrdeutig	ambiguous
Mehrdeutigkeiten auflösen	disambiguate
mehrfach benutzbar	sharable
minor Nummer (Gerätenummer)	minor number
Mitteleuropäische Zeit	MEZ
Modul	module
Modus	mode
Muster	pattern
Neue Zeile	new line
Nicht-Terminalsymbol	nonterminal symbol
Nicht-Terminalzeichen	nonterminal symbol
Nullbyte	null byte (0)
Nullzeiger	null pointer
Objektcode	object code
Objektdatei	object program, object file
Objektmodul	objectprogram
Offset	offset
oktal (8)	octal
Operatorzeichen	operator, operator character
optimieren	optimize
Optimierer	optimizer
Originaldatei	initial delta
Parameter	flag
Parser	parser
Parsergenerator	compiler-compiler
passen	match
Pfad	path
Pfadname	pathname
Phasenfehler	phase error
Pipe, Einwegkanal	pipe
portierbar	portable
Portierbarkeit	portability
Position	position

Präprozessor	preprozessor
Präzedenz	precedence
Priorität	priority
Prozeß	process
Prozeß wieder aufsetzen	resume execution
Prozeß-Dateistatus-Byte	file descriptor flag
Prozeßbeendigung	process termination
Prozeßgruppe	process group
Prozeßgruppenchef	process group leader
Prozeßgruppennummer	process grouppid
Prozeßmaske	process's mode mask
Prozeßnummer	process ID (PID)
Prozeßnummer des Vaters	parent process ID (PPID)
Prozeßspezifische Daten im System	USER area
(auf Konsistenz) prüfen	validate
Prüfprogramm	checker, verifier
Prüfsumme	check-sum
Puffer	buffer
Quellcode	source code
Quell-Programm	source program
reale Benutzernummer	real UID
reale Gruppennummer	real GID
reduzieren	reduce
Regel	rule
Regelteil	rule section
Register	register
regulärer Ausdruck	regular expression
rekursiv	recursive
Release	release
Releasenummer	release
Relokationsbit	relocation bit
Root-Dateiverzeichnis	root directory
Rückkehr	return
s-Bit(Eigentümer,Gruppe)	setuid-Bit (s-Bit)
sbe-Bit (schließe bei exec)	close-on-exec Byte
Scanner	lexical analyzer, scanner
Schalter	flag
Schleife	loop
Schlüssel	key
Schrägstrich	slash
schreiben	write
Schreiberlaubnis	write permission
Schutzbit	protection bit
Segment	segment
Sequence	sequence
serielle Schnittstelle	asynchron communication
Shell	shell
Signal	signal

Sohnprozeß	child process
Sonderzeichen	special character
Speicherabzug	core, core image, core dump, corefile
Speicherplatz	memory
sperren	lock
Stackrahmen	stack frame
Stacksegment	stack segment
Standard-Fehlerausgabe	standarderror
Standardausgabe	standard output
Standardeingabe	standard input
Standardwert	default value
Startadresse	entry point
Startsymbol	start symbol
statisch	static
statischer Datenbereich	static data area
steuern	schedule
Struktur	structure
Swapbereich	swap area
Symboltabelle	symbol table, name-list
syntaktische Variable	nonterminal symbol
Syntaxfehler	syntax error
System	system
System-Dateistatus-Byte	file flag
System zur Verwaltung und Entwicklung von Textdateien	SCCS, Source Code Control System
Systemaufruf	system call
Systembenutzer	user
Systemkern	kernel
Systemprozeß	special process
Systemzeit	system time
Sytemverwalter	super user
t-Bit	sticky-Bit
Tabulatorzeichen	tab
tags-Datei	tags file
Taste	key
Terminalsymbol	token, terminal symbol
Terminalzeichen	token, terminal symbol
Testhilfe	debugger
Textmuster	scanning pattern
Textsegment	text image, text segment
Textsymbol	text symbol
Textzeichen	text character
Token	token, terminal symbol
Tokennummer	token number
Topdelta	top delta
topologisches Sortieren	topologocal sort
Trace Flag	trace flag
Treiber	driver

Übergang	transition
Übergangstabelle	parsing table
Überlauf	overflow
Übersetzer	compiler
Übersetzungstabelle	translation table
Umgebung	environment
Umgebungsvariable	environmental variables
Umleitung	redirection
undefiniert	undefined
Unterbrechung	interrupt
Unterstrich	underscore
Variable	variable
Vaterprozeß	parent process
Verkettung	concatenation
Version	delta, version
Versionsnummer	SID
Verweis	link
Vorausschau	lookahead
Vorrang	precedence
Vorzeichen	sign
Warteschlange	queue (FIFO)
Wert	value
Zeichen	character
zeichenorientiertes Gerät	character special device
Zeichenreihe	string
Zeiger	pointer
Zeittabelle	profile data
Zeitzone	timezone
Zentraleinheit	CPU
Ziffer	digit
Zufallszahl	random number
Zugriff	access
Zugriffsart	access mode
Zugriffsberechtigung	access permission
Zustand	state
Zuweisung	assignment
Zweig	branch
Zweignummer	branch
zweistelliger Operator	binary operator
Zwischendatei	temporary file
Zwischenraum	space

Fachwörter englisch - deutsch

absolute symbol	absolutes Symbol
accept	akzeptieren
access	Zugriff
access mode	Zugriffsart
access permission	Zugriffsberechtigung
action	Aktion
address	Adresse
address space	Adreßraum
alarm clock	Alarmuhr
ambiguous	mehrdeutig
archive	Bibliothek
argument	Argument
array	Feld, Vektor
assembler	Assembler
assembly source program	Assembler-Quellcode, Assembler-Quellprogramm
assignment	Zuweisung
associative	assoziativ
asynchron communication	serielle Schnittstelle
automatic	automatisch
automaton	Automat
base address	Basisadresse
back space key	Korrekturtaste
backslash	Gegenschrägstrich
binary file	Binärdatei
binary operator	zweistelliger Operator
blank	Leerzeichen
block special device	blockorientiertes Gerät
branch	Zweig, Branch, Zweignummer
break	break, break-Anweisung
breakpoint	Haltepunkt
bss	Datensegment, bss-Segment, nicht initialisierte Daten
buffer	Puffer
buffered I/O	gepufferte Ein/Ausgabe
bug	Fehler
byte	Byte = 8 Bit
call	aufrufen
cast	cast, cast-Anweisung
catch	abfangen (Signal)
character	Zeichen
change	ändern
character special device	zeichenorientiertes Gerät
checker	Prüfprogramm
check-sum	Prüfsumme

child process	Sohnprozeß
cleanup	Bereinigung
close-on-exec Byte	sbe-Bit (schließe bei exec)
common symbol	common-Symbol
compiler	Übersetzer
compiler-compiler	Parsergenerator
concatenation	Verkettung
console	Konsole
context-free	kontextfrei
context sensitivity	Kontextabhängigkeit
control character	Kontrollzeichen
control statement	Kontrollanweisung
core	Speicherabzug
core dump	Speicherabzug
corefile	Speicherabzug
core image	Speicherabzug
CPU	Zentraleinheit
current	aktuell
current environment	aktuelles Umfeld
data area	Datensegment
data segment	Datensegment
data symbol	Datensymbol
debugger	Programm zur Fehleranalyse, Testhilfe
debugging	Fehlersuche
decimal	dezimal (10)
default value	Standardwert
delete	löschen
delta	Version, Delta
descriptive text	beschreibender Text
deterministic	deterministisch
device	Gerät, externer Datenträger
digit	Ziffer
directory	Dateiverzeichnis
disambiguate	Mehrdeutigkeiten auflösen
disk	Festplatte
double floating point	doppelte Gleitpunktzahl
driver	Treiber
dual	dual (2)
dummy	Dummy
dynamic	dynamisch
effective GID	effektive Gruppennummer
effective PID	effektive Prozeßnummer
effective UID	effektive Benutzernummer
elapsed time	Laufzeit
end-of-file	Dateiende
entry point	Startadresse
environment	Umgebung
environmental variables	Umgebungsvariable

error	Fehler
error code	Fehlercode
error diagnostic	Fehlermeldung
error handling	Fehlerbehandlung
escape	Flucht
executable program	ablauffähiges Programm
execute	ausführen, durchsuchen
execute permission	Ausführberechtigung
execution	Ausführung
exit status	Endestatus
expression	Ausdruck
external reference	externe Referenz
external symbol	externes Symbol
file	Datei
file address	Dateiadresse
file descriptor	Dateikennzahl
file descriptor flag	Prozeß-Dateistatus-Byte
file flag	System-Dateistatus-Byte
file name	Dateiname
file pointer	Dateizeiger
file status	Dateistatus
FILE structure	FILE-Struktur
file system	Dateisystem
finite automaton	endlicher Automat
flag	Anzeiger, Schalter, Parameter
floating point number	Gleitkommazahl, Gleitpunktzahl
floppy disk	Diskette
format	Format
formated I/O	formatierte Ein/Ausgabe
fractional part	Bruchteil
free	freigeben
GMT	Greenwich Meantime
global	global
grammar	Grammatik
group	Gruppe
group ID (GID)	Gruppennummer
groupname	Gruppenname
hexadecimal	hexadezimal (16)
home directory	Home-Dateiverzeichnis
id keyword	Identifikations-Schlüsselwort
identification keyword	Identifikations-Schlüsselwort
include file	include-Datei
increment	Inkrement
initial delta	Originaldatei
initialize	initialisieren
inode	Indexeintrag
input	Eingabe
input stream	Eingabestrom
insert	einfügen

instruction	Befehl
interrupt	Unterbrechung
inumber	Indexnummer
kernel	Systemkern
key	Schlüssel, Taste
leaf	Blatt
level	Level, Levelnummer
lexical analysis	lexikalische Analyse
lexical analyzer	Scanner
library	Bibliothek
link	Verweis
loader	Binder
local	lokal, nicht global
local directory	aktuelles Dateiverzeichnis
lock	sperren
login name	login-Name
lookahead	Vorausschau
loop	Schleife
LR(1) parsing algorithm	LR(1)-Parsing-Algorithmus
macro	Makro
magic number	magic number
major number	major Nummer (Gerätetyp)
map	Abbildung
mask	Maske
match	passen
memory	Speicherplatz
MEZ	Mitteleuropäische Zeit
minor number	minor Nummer (Gerätenummer)
mode	Modus
modification request	Änderungsgrund, Änderungswunsch
monadic operator	einstelliger Operator
module	Module
mount	einhängen (Dateisystem)
MR	Änderungsgrund, Änderungswunsch
name-list	Symboltabelle
new line	Neue Zeile
node	Knoten
nonterminal symbol	Nicht-Terminalsymbol, Nicht-Terminalzeichen, syntaktische Variable
null byte (\0)	Nullbyte
null pointer	Nullzeiger
object code	Objektcode, assembliertes Programm
object file	Objektdatei
object program	Objektmodul, Objektdatei
octal	oktal (8)

offset	Offset, Abstand
operator	Operator, Operatorzeichen
operator character	Operatorzeichen
optimize	optimieren
optimizer	Optimierer
output	Ausgabe
overflow	Überlauf
owner	Eigentümer
parent process	Vaterprozeß
parent process ID (PPID)	Prozeßnummer des Vaters
parser	Parser
parsing table	Übergangstabelle
pass	Lauf
patch	ausbessern
path	Pfad
pathname	Pfadname
pattern	Muster
pipe	Pipe, Einwegkanal
phase error	Laufzeitinkonsistenz, Phasenfehler
pointer	Zeiger
portable	portierbar
portability	Portierbarkeit
position	Position
preprozessor	Präprozessor
priority	Priorität
process	Prozeß
process group	Prozeßgruppe
process group leader	Prozeßgruppenchef
process grouppid	Prozeßgruppennummer
process ID (PID)	Prozeßnummer
process termination	Prozeßbeendigung
process's mode mask	Prozeßmaske
profile data	Zeittabelle
prompt	Bereitzeichen
protection bit	Schutzbit
queue (FIFO)	Warteschlange
random number	Zufallszahl
read	lesen
read permission	Leseerlaubnis
real GID	reale Gruppennummer
real PID	reale Prozeßnummer
real UID	reale Benutzernummer
recursive	rekursiv
redirection	Umleitung
register	Register
regular expression	regulärer Ausdruck
release	Release, Releasenummer
relocation bit	Relokationsbit

request	Anfrage
result	Ergebnis
resume execution	Prozeß wieder aufsetzen
return	Rückkehr
root directory	Root-Dateiverzeichnis
rule	Regel
rule section	Regelteil
runtime startoff	Laufzeit-Startfunktion
scanner	Scanner
scanning pattern	Textmuster
SCCS	Source Code Control System, System zur Verwaltung und Entwicklung von Textdateien, SCCS
schedule	steuern
screen	Bildschirm
segment	Segment
sequence	Folgenummer, Sequence, laufende Nummer
setuid-Bit (s-Bit)	s-Bit (Eigentümer,Gruppe)
sharable	mehrfach benutzbar
shell	Shell
shift	lesen
SID	Versionsnummer, Deltanummer
sign	Vorzeichen
signal	Signal
size	Größe
slash	Schrägstrich
source code	Quellcode
source programm	Quell-Programm
space	Zwischenraum
special character	Sonderzeichen
special file	Gerätedatei
special process	Systemprozeß
stack (LIFO)	Keller
stack frame	Stack-Rahmen
stack segment	Stacksegment
standard input	Standardeingabe
standard output	Standardausgabe
standarderror	Standard-Fehlerausgabe
start symbol	Startsymbol
state	Zustand
static	statisch
static data area	statischer Datenbereich
sticky-Bit	t-Bit
stream	Datei mit FILE-Struktur und Dateizeiger
string	Zeichenreihe
structure	Struktur

super user	Sytemverwalter
swap	austauschen
swap area	Swapbereich
symbol table	Symboltabelle
syntax error	Syntaxfehler
system	System
system call	Systemaufruf
system time	Systemzeit
tab	Tabulatorzeichen
table of contents	Inhaltsverzeichnis
tags file	tags-Datei
temporary file	Zwischendatei, temporäre Datei
terminal	Bildschirm, Datensichtstation
terminal symbol	Token, Terminalzeichen, Terminalsymbol
text character	Textzeichen
text image	Textsegment
text segment	Textsegment
text symbol	Textsymbol
timezone	Zeitzone
token	Token, Terminalzeichen, Terminalsymbol
token number	Tokennummer
top delta	Topdelta
topological sort	topologisches Sortieren
trace	Ablauf verfolgen
trace flag	Trace Flag
transition	Übergang
translation table	Übersetzungstabelle
tree	Baum
tty group	Bildschirmgruppe
umount	aushängen (Dateisystem)
undefined	undefiniert
underscore	Unterstrich
user	Systembenutzer
USER area	Prozeßspezifische Daten im System
user ID (UID)	Benutzernummer
user time	Benutzerzeit
validate	(auf Konsistenz) prüfen
value	Wert
variable	Variable
verifier	Prüfprogramm
version	Version, Delta
working directory	aktuelles Dateiverzeichnis
write	schreiben
write permission	Schreiberlaubnis

Literatur

/1/ CES - Buch 2
Grundlagen und Kommandos
Bestellnummer:

/2/ Betriebssystem SINIX, Buch 1
Bestellnummer: U1901-J-Z95-1

/3/ UNIX Time-sharing-System,
UNIX Programmer's Manual Vol. 1-3
Seventh Edition,
Murray Hill: Bell Telephone Laboratories 1979

/4/ X/OPEN Portability Guide
Elsevier Science Publishing company New York, 1985

/5/ The Bell System Technical Journal
Volume 57, No. 6, Part 2
American Telephone and Telegraph Company, 1978

/6/ Jürgen Gulbins
UNIX
Springer-Verlag Berlin, 1984

/7/ M.Banahan, A. Rutter
UNIX- lernen, verstehen, anwenden
Hanser Verlag München, 1984
Deutsche Ausgabe von Prof Dr.A.T.Schreiner, T.Mandry

/8/ B.W. Kernighan, D.M. Ritchie
Programmieren in C
Hanser Verlag München, 1983
deutsche Ausgabe von Prof Dr. A. T.Schreiner,
Dr. Ernst Janich

/9/ Jack Purdom
Einführung in C
Markt&Technik- Verlag München, 1983
Deutsche Übersetzung Peter Lüke

/10/ Kaare Christian
The UNIX operating system
John Wiley&Sons, 1983

/11/ W.Kernighan, R.Pike
The UNIX Programming environment
Prentice-Hall, 1984

/12/ S.R.Bourne
The UNIX System
Addison-Wesley Publishing Company 1983

/13/ E.Allman
An Indroduction to the Source Code Control System
Project Ingres
University of California at Berkeley

/14/ L.E.Bonanni, C.A.Salemi
Source Code Control System
User's Guide
Bell Telephone Laboratories Incorporated

Stichwörter