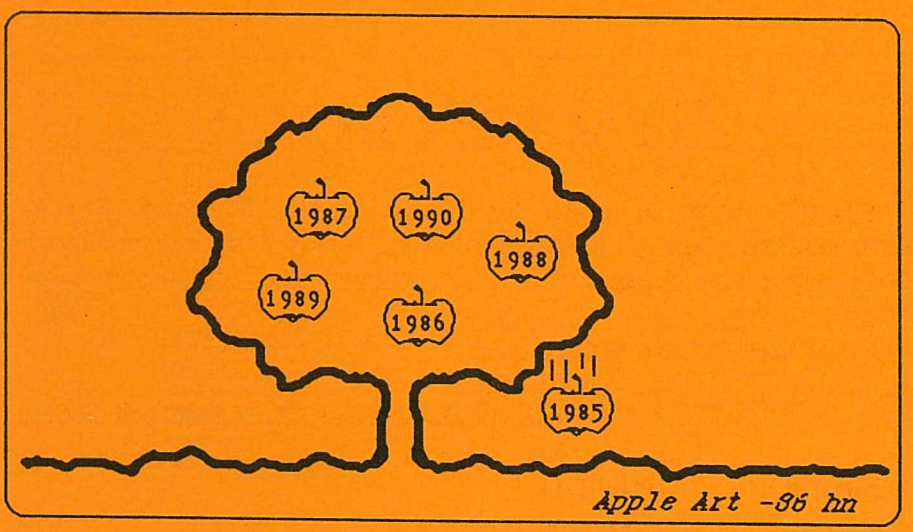
StackPointer 1-1986



Datorföreningen STACKEN

Datorföreningen STACKEN är en kårförening med omkring 200 medlemmar. Föreningen har funnits sedan 1978.

Genom föreningen skall man kunna få utlopp för sitt intresse för datorer. Den skall också verka för erfarenhetsutbyte mellan medlemmarna och vara ett gemensamt forum.

Vilka aktiviteter vi har beror på vad medlemmarna hittar på. Vi gör studiebesök hos olika leverantörer och installationer. Vi har utbyte med andra datorföreningar i Norden. Vi hittar föreläsare, som delar med sig av sina erfarenheter. Vi odnar filmvisning. Vi har en DEC-10-dator, som vi har installerat och satt igång.

Den första helgfria torsdagen varje månad, klockan 19, träffas vi i lokalen vid datorn. Kom på ett möte, om du är intresserad. Vi tar gärna emot fler medlemmar. Medlemsskap i STACKEN kan beviljas efter ansökan till föreningens styrelse. Medlemsavgiften är 86 kronor för 1986.

Adress:	c/o NADA KTH 100 44 STOCKHOLM
Besöksadress:	Brinellvägen 32 (på gaveln mot Lill-Jansskogen)
Postgiro:	433 01 15-9
Bankgiro:	344-3595

STACKPOINTER

STACKPOINTER är organ för Datorföreningen STACKEN. STACKPOINTER utkommer när material i tillräcklig mängd finns, förhoppningsvis 5–6 gånger per år. Återgivande av delar av innehållet är tillåtet om källan anges.

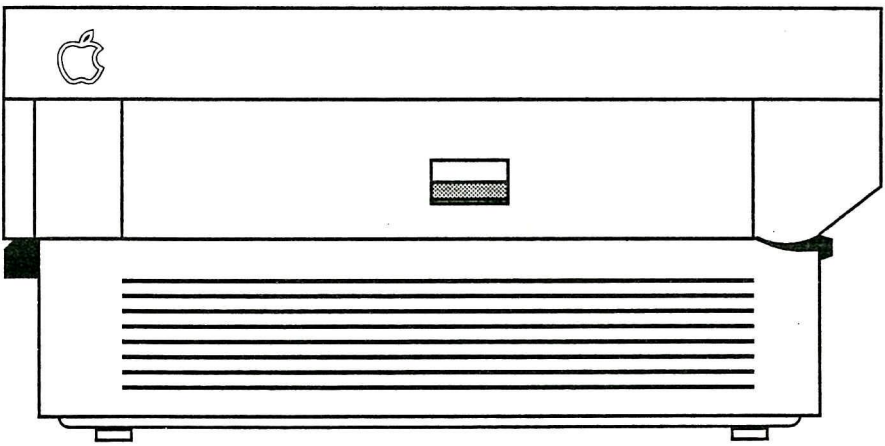
Ansvarig utgivare:	Mats O Jansson
Redaktör:	Hans Nordström
I redaktionen:	Jan Michael Rynning, Mats O Jansson och Stellan Lagerström
Färdigställd:	1986-01-14

Laserskrivaren som både kan skriva och räkna

Apples laserskrivare har en inbyggd interpretator för ett FORTH-liknande språk, som heter PostScript. Om man vill skriva ut någonting på den skickar man över det som ett PostScript-program, som den får exekvera.

Jan Michael Rynning

```
% PostScript-program för framsidan på StackPointer 1-1986.
% Jan Michael Rynning, 1986-01-05.

% Placera origo mitt på pappret.
320 370 translate

% Definiera storlek på bokstäverna och välj typsnitt.
/large-size 35 def
/Helvetica-Bold findfont large-size scalefont setfont

% Definiera två funktioner för att skriva ut roterande text.
/rotate-page ä  % Anrop: sträng  rotate-page  sträng
  165 -15 0 ärotate-lineå for
å def
/rotate-line ä  % Anrop: sträng vinkel  rotate-line  sträng
  gsave
    rotate dup stringwidth pop -2 div large-size -0.35 mul moveto
    gsave 1 setgray dup show grestore
    gsave dup false charpath 0.5 setlinewidth stroke grestore
  grestore
å def

% Skriv ut roterande "StackPointer 1-1986".
(StackPointer 1-1986) rotate-page pop

% Mata ut sidan.
showpage
```



Kallelse till vårmöte

Härmed kallas till vårmöte i Datorföreningen STACKEN. Mötet kommer att hållas i B30, STACKENs lokal på Brinellvägen 30, Kungliga Tekniska Högskolan, med början klockan 1900 torsdagen 1986-02-06.

Förslag till dagordning:

§1. Mötets öppnande.
§2. Val av justeringsmän.
§3. Val av mötesordförande.
§4. Val av mötessekreterare.
§5. Tillkännagivande av uppgjord röstlängd.
§6. Fråga om mötet är stadgeenligt utlyst.
§7. Fråga om dagordningens godkännande.
§8. Verksamhetsberättelse.
§9. Revisionsberättelse.
§10. Balansräkning.
§11. Ansvarsfrihet för avgående styrelse.
§12. Ev. val av styrelsemedlemmar.
§13. Ev. fastställande av firmatecknare.
§14. Ev. val av revisorer.
§15. Ev. val av valberedning.
§16. Mötesdagar under 1986.
§17. Motioner.
§18. Övriga frågor.
§19. Mötets avslutande.

För styrelsen, *Mats O Jansson*

Motion angående hedersmedlemskap.

Jag anser att STACKEN skall ge JMR hedersmedlemskap för sina insatser som ordförande under två år.

Stockholm 860107 1530, *Peter Löthberg*

Rebus

$\approx 3141592653589.793$

Lösning på åldersproblem

Den mest oväntade lösningen på åldersproblemet i STACKPOINTER 6-1985 kommer från Lennart Börjesson, Undervisningsdatorn Linje E. Han har använt **METAFONT** för att lösa ekvationerna. **METAFONT** är ett programspråk, som har tagits fram av Professor Donald E Knuth på Stanforduniversitetet och är främst avsett för att konstruera typsnitt. Knuth har bl a använt det för att göra Computer Modern-typsnitten, som vi använder i STACKPOINTER. När man definierar hur ett tecken i ett typsnitt ska se ut, med hjälp av ett antal relationer, så är det enklast om man kan skriva dem i vilken ordning som helst, och inte nödvändigtvis i den ordning datorn räknar ut dem. Knuth har därför försett **METAFONT** med förmågan att lösa linjära ekvationssystem, vilket kom väl till pass här.

Jan Michael Rynning

```
Elmer>MF
This is METAFONT, Version 0.81 for Vax/VMS (no base preloaded)

*A+B+C+D+E=150;
*A=C+D;
*B=C+E;
*C=D+E;
*D=B/2;
*E=D/2;
! Redundant equation.
<to be read again>
                   ;
<*> E=D/2;

?
*A+D=B+C;
! Redundant equation.
<to be read again>
                   ;
<*> A+D=B+C;

?
*show A;
>> 50
*show B;
>> 40
*show C;
>> 30
*show D;
>> 20
*show E;
>> 10
*end;
Transcript written on ELMER1:ÄLENNARTBÅMFPUT.LIS;2.
Elmer>
```

AMIS på Katia

Fyra dagar efter Peter hade fått igång TOPS-10 på Katia nere i CCCC, så tyckte jag det var dags att försöka få en fungerande AMIS till henne. STACKENs egen editor på STACKENs egen dator ...

Eftersom jag visste att det skulle krävas en del ändringar i AMIS, så skickade jag ett KOM-brev till Johnny och bad honom kopiera över källkoden från Oden till Kicki. Åkte hem och sov. Vaknade, åkte tillbaka till CCCC. Ingen källkod.

Jag hade ingen lust att vänta, så jag bestämde mig för att gå den hårda vägen — patcha direkt i maskinkoden. Dessutom såg jag det som en rolig utmaning att få igång AMIS på det sättet.

Tre olika typer av problem väntade mig:

Katia har en KA10-processor, som inte klarar av alla de maskininstruktioner som finns på de modernare KI10-, KL10- och KS10-processorerna. Det problemet hade jag redan löst, genom att skriva en emulator för DMOVE och DMOVEM — de två mest använda av de instruktioner som inte finns på KA10.

Minneshanteringen på KA10 skiljer sig från den på de andra processorerna. KA10 delar in användarprogrammets adressrymd i två segment: lågsegmentet, som börjar på adress 0_8, och högsegmentet, som börjar mitt i adressrymden, på adress 400000_8. För vart och ett av segmenten finns ett bas- och ett längdregister. Basregistren pekar ut var i det fysiska minnet segmenten börjar och längdregistren är till för att hindra användarprogrammet från att läsa och skriva i minnesadresser utanför segmenten.

På de modernare processorerna består den logiska adressrymden av 512 sidor à 512 ord. Översättningen från logiska till fysiska adresser görs med hjälp av en tabell, som anger var i det fysiska minnet de olika sidorna börjar. TOPS-10 simulerar indelningen i låg- och högsegment genom att lägga in lämpliga värden i tabellen, men högsegmentet behöver inte börja på 400000_8.

AMIS högsegment börjar på 600000_8, för att man ska kunna ha ett större lågsegment och därmed kunna editera större filer. För att kunna köra AMIS på Katia var jag tvungen att flytta ner högsegmentet till 400000_8. Det gjorde jag genom att skriva ett litet PASCAL-program, som gick igenom AMIS.EXE ord för ord, kollade om värdet i höger halvord kunde vara en adress i högsegmentet och isåfall subtraherade 200000_8 från den. Jag kunde inte vara säker på att

allt jag ändrade var adresser — det kunde ju vara texter eller någonting annat — men det var värt ett försök. Till sist patchade jag EXE-filsdirectoryt med FILDDT, så monitorn skulle ladda in högsegmentet på rätt adress.

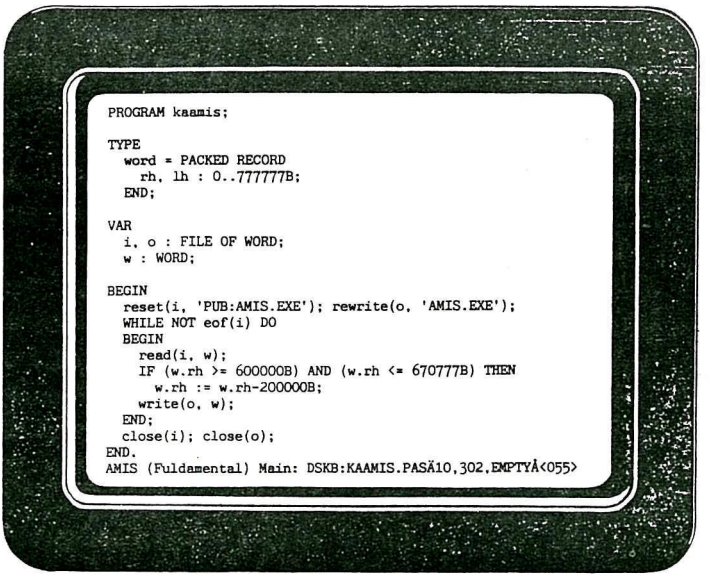
Dags att provköra. "?Illegal memory reference". En tabell med adresser i vänster halvord visade sig vara boven. Fixade till den med FILDDT. Nu gick den patchade AMIS-versionen på Kicki — fast en del texter hade fått stryk. Det stod "Fuldamental" i st f "Fundamental" på modraden, t ex.

Återstod att flytta över AMIS till Katia och övervinna det tredje hindret. Den modernaste version av TOPS-10 som går att köra på KA10 är 6.03A. I senare versioner av operativsystemet har det tillkommit en del saker, bl a några terminalparametrar som AMIS försöker läsa av.

Första försöket resulterade i att AMIS skrev ut att dittan-dattan terminalparameter inte gick att läsa av och gjorde EXIT. Grrr! Morrr! Frässs! Fel som man inte kan göra någonting åt och som inte har någon betydelse ska man ignorera! I synnerhet som det som är fel är rätt i äldre versioner av operativsystemet. Felkontrollerna försvann, en efter en, tills AMIS gav med sig och lät sig köras på Katia.

Och än i dag körs denna version av AMIS på Katia, fast Hazze Slöberg har ändrat tillbaka till "Fundamental".

Jan Michael Rynning



```
PROGRAM kaamis;

TYPE
  word = PACKED RECORD
    rh, lh : 0..777777B;
  END;

VAR
  i, o : FILE OF WORD;
  w : WORD;

BEGIN
  reset(i, 'PUB:AMIS.EXE'); rewrite(o, 'AMIS.EXE');
  WHILE NOT eof(i) DO
  BEGIN
    read(i, w);
    IF (w.rh >= 600000B) AND (w.rh <= 670777B) THEN
      w.rh := w.rh-200000B;
    write(o, w);
  END;
  close(i); close(o);
END.
AMIS (Fuldamental) Main: DSKB:KAAMIS.PASÄ10,302,EMPTYÅ<055>
```

DATE-86

Early this year a message appeared on ARPANET-BBOARDS commemorating the ten-year anniversary of DATE-75. A somewhat more ominous anniversary will occur in four weeks, on 9 January 1986. Users of the TOPS-10 operating system should beware of software failures beginning on that date.

DATE-75 is the name of a set of program modifications applied to the TOPS-10 operating system, running on DEC PDP-10 computers. Before the modifications, the TOPS-10 system could only represent dates between 1 January 1964 and 4 January 1975. The DATE-75 modifications added three more bits to the representation of dates, so that dates up to 1 February 2052 could be represented. To maximize compatibility with existing software, the three extra bits were taken from several unused positions in existing data structures. The change was announced in mid-1974, and several tens of person-years went into updating software to recognize the new dates.

Unfortunately, reassembling these bits into an integer representing the date was somewhat tricky. Also, some programs had already used the spare bits for other purposes. There were a large number of bugs that surfaced on 5 January 1975, the first day whose representation required the DATE-75 modification. Many programs ignored or cleared the new bits, and thought that the date was 1 January 1964. Other programs interpreted the new bits incorrectly, and reported dates in 1986 or later. Date-related program bugs were frequent well into the Spring of 1975.

On 9 January 1986, the second bit of the DATE-75 extension will come into use. Users of software developed in the 60's and early 70's on the TOPS-10 operating system should beware of problems with testing and manipulation of dates. Beware especially of programs that were patched after manifesting bugs in 1975, for in the rush to fix the bugs it is possible that some programs were modified to assume that the date was between 1975 and 1986. Any date that is off by a multiple of eleven years and four days is probably caused by this type of bug.

Dan Hoey

```
CHKD75: SKIPG   S.DT75##        ;SEE IF /DATE75
        CAIE    T4,1            ;NO--IF 1,
        SKIPA                   ;ELSE
        MOVEI   T4,0            ;IF NOT /DATE75 AND LOST, SET 0

        CAIE    T4,1            ;UNLESS JUST DATE LOSAGE,
        JRST    CHKLMX          ; GO RETURN
        MOVEI   T4,0            ;POSSIBLE DATE75, SET FOR FAILURE
        HLRZ    T1,.CCDATI      ;GET CREATION DATE
        CAIL    T1,115103       ;IF BEFORE 1-JAN-67
        CAIN    T1,122661       ; OR = 5-JAN-75
        MOVEI   T4,1            ;INDICATE DATE75

        HLRZ    T1,.CADATI      ;GET ACCESS DATE
        CAIL    T1,115103       ;IF BEFORE 1-JAN-67
        CAIN    T1,122661       ; OR = 5-JAN-75
        MOVEI   T4,1            ;INDICATE DATE75
```

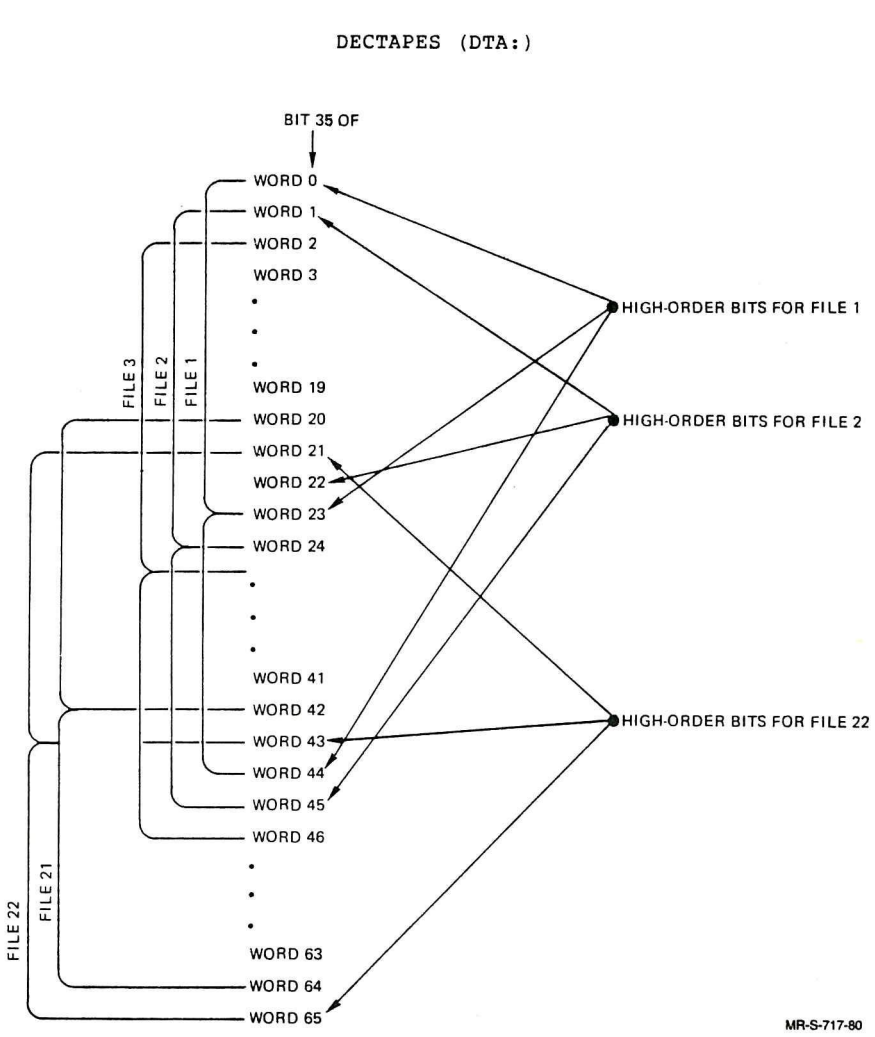


Figure 15-6 High-Order Three Bits of Creation Date

15.4.1.4 **File Creation Dates** - The low-order 12 bits of the creation date for each file is in bits 24 through 35 of the same word containing the extension for the file. The high-order 3 bits of the creation dates are stored in the last bit of words 0 through 82 of the directory (see Figure 15-6).

Så göra vi när vi laga våran KA

Onsdagen 13 Dec skulle det vara hårdvarukurs på Katia. Problemet var att hon inte ville. Vissa instruktioner fick processorn att hänga sig. Någon händig person trodde att det skulle nog fixa sig om man ruskade på CPU-skåpet. Därefter gick ingenting.

Sen stod Katia tills i helgen.

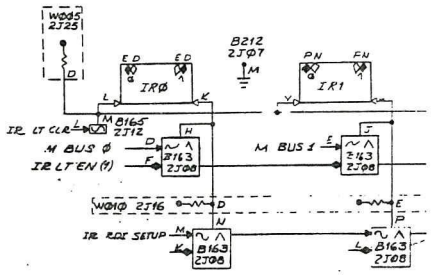
Vi börjar med att titta på lamporna och trycka på knapparna. Alla bitar i vänster halva av instruktionsregistret (IR) är satta, vilken instruktion vi än försöker exekvera. Adressberäkningen fungerar dock.

Vi tittar i kopplingsschemat. Konstiga symboler. Vi läser KA10 Maintenance Manual. Länst bak fanns en appendix om hur man läste schemorna. Mhmmm, jaha, på så sätt... är det *det* de där konstiga krumelurerna är! Konstig djävla maskin; ingen mikro-PC, bara timing-pulser som springer omkring i logiken, dvs, om de försvinner ngn stans, så hänger sig processorn...

I schemat över IR finns en ledning som heter IR LT CLR. Det borde betyda IR Left Clear enligt boken. Den har hand om just de bitar det gäller. Kanske försvinner den signalen ngn stans?

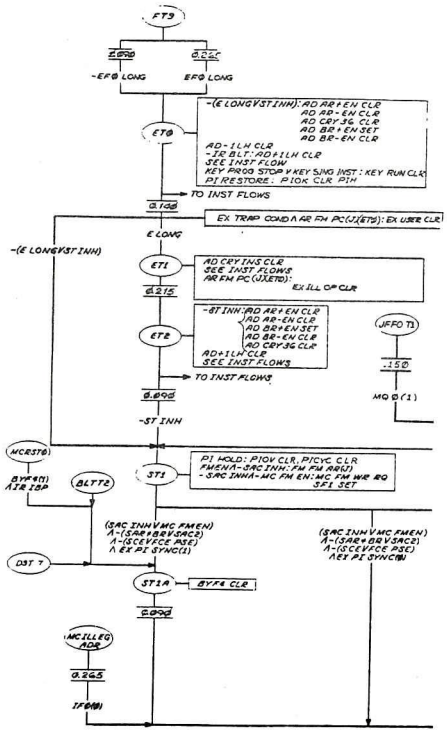
Vi stänger av strömmen och sätter på den igen. Nu är bara några bitar satta. Det visar sig att så fort en bit blivit satt, så är den alltid det sen.

Vi satte frontpanelen på att repetera en instruktion och letade efter signalen i bakplanet. IR LT CLR, ska finnas på kort 2J12 (skåp 2 rad J kort 12) pinne L. Inte ett spår av signal. Hmmm. Ska komma från 1M40 pinne D. Jaha, där finns den! I skåp 1. Det kanske är kabeln mellan skåpen?



Vi tittar i schemat "inter-bay cables". Där går den! Mellan 1L42 och 2J03. Vi tar ut kabeln och tittar på den (*Efter* att ha stängt av strömmen!) Ser ok ut. Mäter igenom alla trådarna med ohmmeter; alla är hela. Fan! Fast kontakterna den satt i ser ju lite trötta ut... Pillar på kontaktfjädrarna med skruvmejsel och sätter i kabeln igen. Nu går den! Nästan.

Det första felet kvarstår, vissa instruktioner hänger sig. Men vilka, och vad har de gemensamt? Efter en timmes testande av olika instruktioner och funderande hittar vi det i "Basic instruction flow". Där står vilka vilka vägar som signalerna går för alla instruktioner. Alla konstiga instr sätter flaggan E LONG i början. Vad gör den då? I flödes-schemat över "Execute and store" syns att då den är satt så har vi en "lång" instruktion, och timing-pulsen går en omväg. Det måste vara här någonstans den dör ut.



Raskt över till kopplingsschemat. Sätter panelen på repetition av konstig instruktion och fram med scopet. 1T27 pinne L, jaha, där är den. Ska komma ut på N, det gör den. Vidare till ET1 (den heter så). Den finns. Men inte ET2. In på 1T29 L , ut på N, nähä, där försvinner den, då har vi hittat boven!

B311 heter modultypen, nån sorts fördröjningslina. Är det kontakterna igen, eller själva modulen? Vi hittar en likadan i ett av de skrotade minnesskåpen, och byter. Nu går den! Nästan.

Det går att boota, men en del av minnet är konstigt. Vi fyller hela minnet med testord från frontpanelen (Repeat+Deposit Next) och läser tillbaka (Repeat+Examine Next).

I 8kord av minnet är alla bitar i vänster halvord 0. Varje skåp är på 64kord. 8 grupper om 8kord×19 bitar i varje halva. Ett kort, således. Men vilket? Det är 4 kort i varje 8k grupp. Ett är själva kärnminnet, de övriga drivkretsar etc.

Det var ingen brist på reservkort pga en viss lastbilstransport, så vi byter allihopa i den gruppen. Nu går det! Allting!

Men vi vill veta vilket kort det var, kan inte kassera alla 4! Byter tillbaka ett i taget tills ett är kvar. Minnet går fortfarande. Kan Sherlock Holmes, så...

Tittar på det. En elektrolytkonding har exploderat och kortslutit nånting. Vi lägger kortet i lådan för undervisningsmaterial, bootar maskinen och skriver raskt ett KOMinlägg om hur duktiga vi varit (JMR, Thord och jag).

Stellan L

ITS FLAP

FLAP: flap a micro-tape

arg 1 Micro-tape number (typically 1-4).

The directory for the micro-tape is written back onto the tape if it is currently in core; the tape is then physically dismounted by running the tape back onto the original reel (thereby making the tape go flap, flap, flap ...). Micro-tapes should not be manually dismounted, for this will cause the directories to get out of phase, messing up the dismounted tape and also the next one to use the drive. The FLAP will fail if any files are still open on the specified drive, or if any one else has the drive assigned to him.
See also the .UDISMT uuo.

.UDISMT ac, micro-tape dismount
;skip if successful

The specified accumulator should contain a micro-tape drive number. If the uuo succeeds, the directory is written back onto the micro-tape and excised from the system's memory. The micro-tape is then physically dismounted (this is known as "flapping" the tape, since when it runs off the reel it goes flap, flap).
Micro-tapes should not be manually dismounted, for this will cause the directories to get out of phase, messing up that micro-tape and the next one to use that drive.
The .UDISMT will fail if any files are still open on the micro-tape, or if someone else has the drive assigned to him.
DDT makes this available via its command :FLAP.
See also the FLAP symbolic system call.

Redaktörens ruta

Katias olika delar

BA10 Kontrollenhet för radskrivare, plotter, kortläsare och -stans.

DC10 Terminalscanner. Innehåller logiken för terminallinjerna. Man kan ha 3 olika hastigheter på terminallinjerna, men det går inte att ställa om med programvara, utan byglas på varje kort. Det finns ingen statussignalering för modemkontroll i vår DC10.

DF10 Datakanal för de enheter (skivminnen och bandstationer) som gör överföringar via kanalbussen med hög hastighet till och från minnet.

DK10 Realtidsklocka, som är kopplad till I/O-bussen.

KA10 Centralenhet.

MF10 Minnesskåp med 64 Kord kärnminne per skåp.

PC09C Hålremsläsare och stans, som är kopplad till I/O-bussen.

RH10 Kontrollenhet för massbussenheter (skivminnen och bandstationer), t ex RP04, RP06, RM03, RM05 och TM78. Kan styra upp till 8 sådana enheter. Får kommandon via I/O-bussen och skickar data till och från minnet via kanalbussen och DF10.

RM03 Skivminne som rymmer 15 Mord. Styrs av en RH10.

RP02 Skivminne som rymmer 5 Mord. Styrs av en RP10 eller RP10C.

RP04 Skivminne som rymmer 20 Mord. Styrs av en RH10.

RP10C Kontrollenhet för RP02- och RP03-skivminnen. Kan styra upp till 4 sådana skivminnen. Får kommandon via I/O-bussen och skickar data till och från minnet via kanalbussen och DF10.

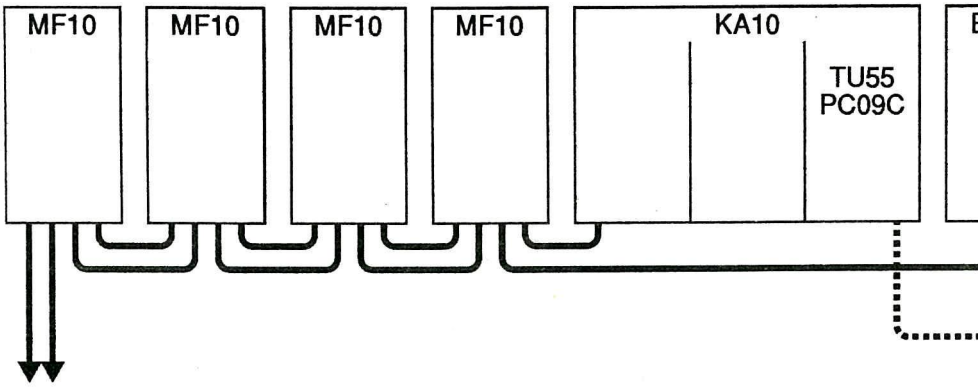
TM10B Kontrollenhet för bandstationer. Kan styra upp till 8 bandstationer.

TU10 9-kanals bandstation. Styrs av en TM10A eller TM10B.

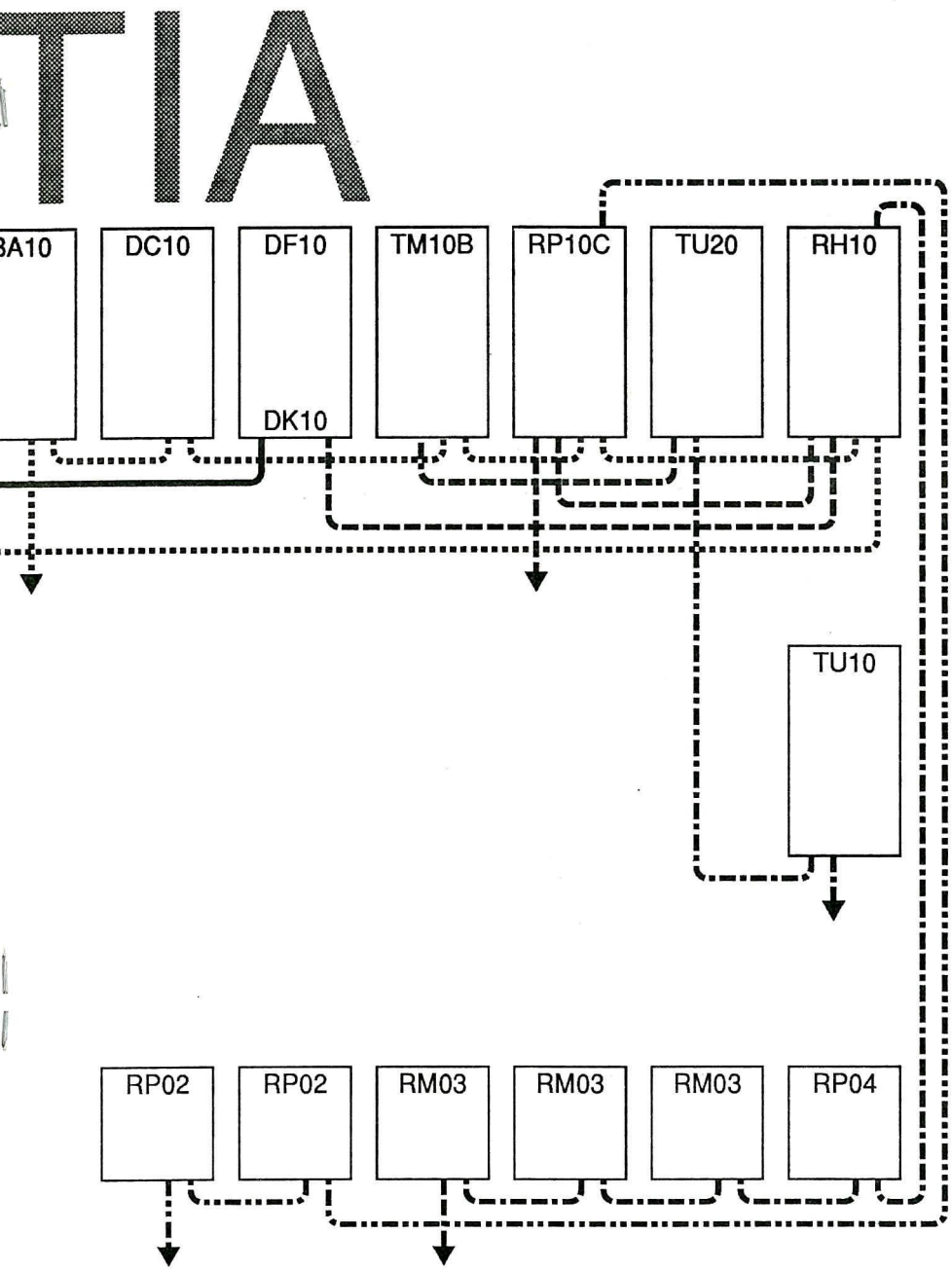
TU20 9-kanals bandstation. Samma som TU10, fast av annat fabrikat.

TU55 DECtape-bandstation. Styrs av en TD10 (om vi hade en).

KA



Minnesbuss
I/O-buss
Kanalbuss
Tapebuss
Diskbuss
Massbuss
Bussavslutning


TIA
3A10
DC10
DF10
DK10
TM10B
RP10C
TU20
RH10
TU10
RP02
RP02
RM03
RM03
RM03
RP04

ZORK

A COMPUTERIZED FANTASY SIMULATION GAME

BY P. DAVID LEBLING
MARC S. BLANK
TIMOTHY A. ANDERSON

MIT LABORATORY FOR COMPUTER SCIENCE

```
Welcome to Zork.
This version created December 8.
West of House
You are in an open field west of a big white house with a
boarded front door.
There is a small mailbox here.
> GO NORTH
North of House
You are facing the north side of a white house. There is
no door here, and all the windows are barred.
> EAST
Behind House
You are behind the white house. In one corner of the
house there is a small window which is slightly ajar.
> OPEN THE WINDOW
With great effort, you open the window far enough to
allow entry.
> GO IN
Kitchen
You are in the kitchen of the white house. A table
seems to have been used recently for the preparation of
food. A passage leads to the west, and a dark staircase
can be seen leading upward. To the east is a small
window which is open.
On the table is an elongated brown sack, smelling of
hot peppers.
A bottle is sitting on the table.
The glass bottle contains:
   A quantity of water
```

Beyond this nondescript kitchen, above and below the surface of the earth, lie scores of rooms, some containing traps, some containing puzzles. Hundreds of objects are scattered throughout this maze, some valuable treasures, some magical tools. The little white house in the forest clearing is the entrance to Zork, a game developed by the authors. Zork is one example

of a new type of game: the Computerized Fantasy Simulation game.

In this type of game, the player interacts conversationally with an omniscient "Master of the Dungeon," who rules on each proposed action and relates the consequences. If the player says "Go north," he may move north, or the dungeon master may say "There is no way to go in that direction." If the player says "Open the window," the dungeon master may respond "The window is locked." The results depend on the design of the game, its architecture and furnishings, so to speak: in one game picking a sword might be fatal; in another it might confer magical powers. The design and implementation of such games is as much an exercise in creative writing as in programming.

The interest in playing Zork (or any other CFS game) is twofold. First, the object of the game is usually to collect treasure, and this may be done only by solving problems; in the above the player would garner 10 points by being clever enough to open the window and enter the house.(Zork itself has more than two dozen distinct problems to solve, some presented in several stages.) Second, a great deal of the enjoyment of such games is derived by probing their responses in a sort of informal Turing test: "I wonder what it will say if I do this?" The players (and designers) delight in clever (or unexpected) responses to otherwise useless actions.

Overview: Simulating the Universe

The heart of any CFS game is its ability to mimic omniscience. By this we mean that the game should simulate the real world sufficiently well so that the player is able to spend most of his time solving the problems rather than solving the program. If, for example, the vocabulary is too small, the player must always wonder if his problem is only that he hasn't yet found the right word to use. Similarly, it is annoying for a possible solution to a problem to be ignored by the game. In other words, the program must simulate the real world.

Obviously, no small computer program can encompass the entire universe. What it can do, however, is simulate enough of the universe to appear to be more intelligent than it really is. This is a successful strategy only because CFS games are goal-directed. As a consequence, most players try to do only a small subset of the things they might choose to do with an object if they really had one in their possession.

Zork "simulates the universe" in an environment containing 191 different "rooms" (places to be) and 211 "objects." The vocabulary includes 908 words, of which 71 are distinct "actions" it handles. By contrast, a person's conversational vocabulary is a factor of two (or more) larger. How, then, does a limited program make a creditable showing in the conversational interaction that characterizes Zork?

The technique Zork uses for simulating the universe is that of universal methods modified for particular situations. For example, when a player tries to take some object, he expects to end up carrying it. There are, as in the real world, exceptions: some objects are "nailed down," and one's carrying capacity is limited. These restrictions are included in the general TAKE function. However, the designer might want a special action in addition to, or instead of, the general TAKE: a knife might give off a blinding light when taken; an attempt to take anything in a temple might be fatal. These exceptions would not appear in the general TAKE function, but in functions associated with the knife and the temple.

The details of this method of exceptions will be taken up later. The effect of it is that "the expected thing" usually happens when the player tries to (for example) take something. If the object he is trying to take is not special, and the situation is not special, then "it works," and he gets the object. In Zork, there are quite a few of these basic verbs. They include "take," "drop," "throw," "attack," "burn," "break," and others. These basic verbs are set up to do reasonable things to every object the player will encounter in the game. In many

cases, objects have properties indicating that a certain verb is meaningful when applied to them (weapons have a "weapon" property, for example, that is checked by the verb "attack"). Applying a verb to an object lacking the necessary property often results in a sarcastic retort. ("Attacking a troll with a newspaper is foolhardy.") But the point is that it does something meaningful, something the player might have expected.

Another way in which the game tries to be real is by the judicious use of assumptions in the dungeon master's command parser. Suppose the player says "Attack." Assuming that he has a weapon and there is an enemy to attack, this should work, and it does. Assumptions are implemented by the existence of verb frames (stereotypes) and memory in the parser. In the example, the parser picks up the verb frames for the verb "attack." They indicate that "Attack 'villain' with 'weapon'" is the expected form of the phrase. Now, "villain" means another denizen of the dungeon, so the parser looks for one that is presently accessible, a "villain" in the same room as the player. Similarly, the player must have a "weapon" in his possession. Assuming only one "villain" is in the room and the player has only one "weapon," they are placed in the empty slots of the frame and the fight is on.

Suppose that there is only one villain available, the troll, but the player has two weapons: a knife and sword. In that case, the dungeon master can't decide for him which to use, so it gives up, saying "Attack troll with what?" However, it remembers the last input, as augmented by the defaults ("Attack troll"). Thus, if the user replies "With sword," or even "Sword," it is merged with the leftover input and again the fight is on. This memory can last for several turns: for example, "Attack"; "Attack troll with what?"; "With knife"; "Which knife?"; "Rusty knife"; and so on.

Data Structure and Control Structure

The underlying structure of Zork consists of the data base

(known as "the dungeon") and the control structure. The data base is a richly interconnected pointer structure joining instances of four major data types: "rooms," locations in the dungeon; "objects," things that may be referenced; "actions," verbs and their frame structures; and "actors," agents of action. Each instance of these data types may contain a function which is the specializing element of that instance. The control structure of Zork is, at one level, a specification of which of these functions is allowed to process an input sentence and in what order.

In the simplest sense, Zork runs in a loop in which three different operations are performed: command input and parsing, command execution, and background processing. (Figure 1 is a flowchart of the Zork program.)

The command input phase of the loop is relatively straightforward. It is intended to let the user type in his command, edit it if he needs to, and terminate it with a carriage return.

The purpose of the Zork parser is to reduce the user's input specification (command) to a small structure containing an "action" and up to two "objects" where necessary.

The parser begins by verifying that all the words in the input sentence are in its vocabulary. Then, it must determine which action and objects, if any, were specified. For an object to be referenced in a sentence, it must be "available"—that is, it must be in the player's possession, in the room the player is in, or within an object that is itself available. Objects not meeting these criteria may still be referenced if they are "global objects," which are of two types: those that are always available (such as parts of the player's body), and those that are available in a restricted set of rooms (such as a forest or a house). Adjectives supplied within the sentence are used to distinguish objects of the same generic type (such as knives and buttons) but are otherwise ignored. If an object remains ambiguous, the parser asks which of the ambiguous objects was meant (for example, "Which button should I push?").

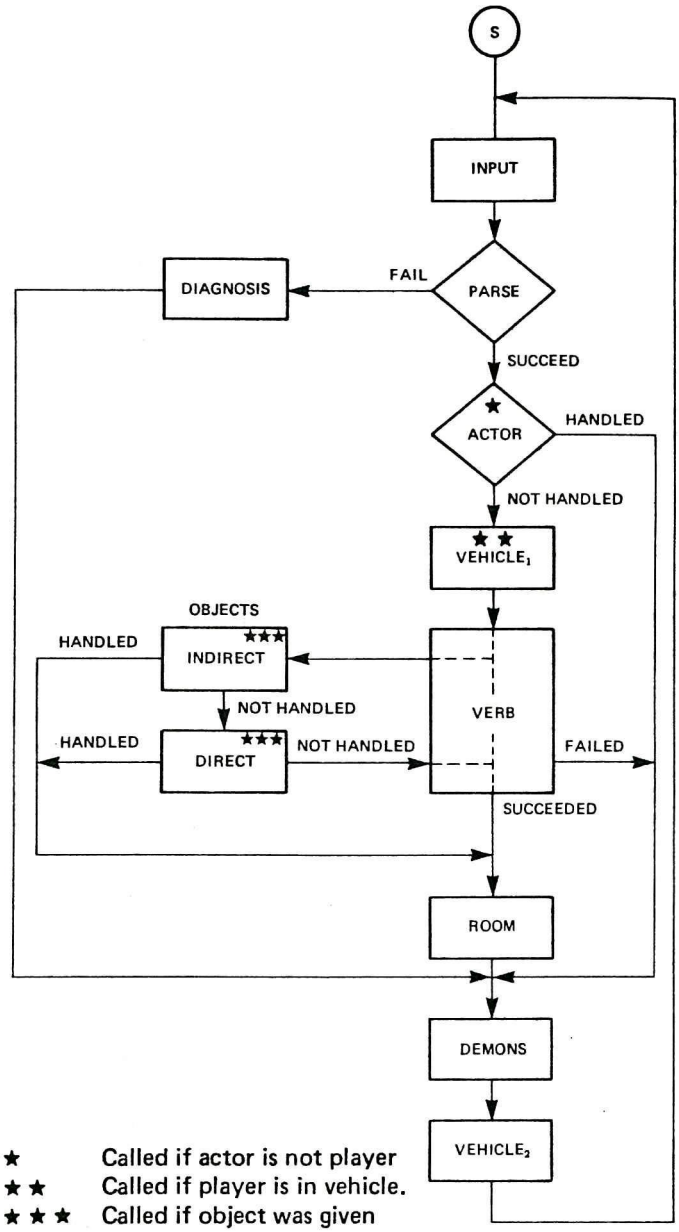


Figure 1. Zork flowchart.

Next is syntax checking, whereby the correct "action" is used for any verb. Syntax checking makes use of any supplied prepositions, differentiating between, for example, "look at" and "look under," which imply different actions. Finally, having determined the appropriate syntax for a given sentence, the parser ensures that all required objects for a given action were specified. The parser may, for example, decide that the correct syntax for the sentence "Give X" is "Give X to Y." An attempt is then made to supply an appropriate "Y" to complete the sentence. This is made possible by the definitions of the actions themselves, which include the attributes of the objects to be acted upon. In the present example, the action for "Give" defines the indirect object ("Y") to be another denizen of the dungeon; the parser attempts to comply by seeing if one is available. If so, the indirect object is found, and the parse is successful. If not, the player is asked to supply the indirect object himself. ("Give X to whom?") Once this phase is completed, the parse is finished and the parser's output is returned.

The adjectives and prepositions that were in the user's input are used only to determine the "action" and the "objects," and are not part of the parser's output. In addition, all articles are ignored, though users may add them to increase the clarity (to themselves) of what they input. For example, an input of "Knock down the thief with the rusty knife" reduces to something like

[<action STRIKE> <object THIEF>
<object RUSTY-KNIFE>]

If, however, the input were "Knock on the thief," the parser would reduce that to

[<action RAP> <object THIEF>]

recognizing that the "action" to be performed depends, for the word "knock," on the syntax of the input sentence:

"knock down" turns into "strike," while "knock on" turns into "rap."

Once parsing has been completed, processing of the command is started. The functional element (if any) of each of the objects in the parsed input may be invoked. Additionally, some objects not specifically referenced, but which define the situation, are part of the processing sequence. The order in which these functions are invoked is determined by a strategy of allowing the exceptions an opportunity to handle the input before performing the action associated with the most general case. The processing order is as follows:

- The actor performing the command, if any. This allows, for example, a robot with a limited range of actions.
- The vehicle the actor is in, if any. This allows the vehicle to restrict movement. For example, inside a freely drifting balloon the normal movement commands (such as "Run north") might be meaningless or even fatal.
- The verb, or "action."
  (a) The indirect object of the sentence, if any.
  (b) The direct object of the sentence, if any.
- The vehicle again, if any. The vehicle is called a second time to enable it to act based on changes in the state resulting from the action.
- The room the player is in.

Each of these functions is invoked in turn and given the option of handling the command. If it decides to handle the command, processing terminates at that point, and the remaining functions are not invoked. Otherwise, the sequence continues. Note that a function may do something (such as print a message) without completely handling the input. The invocation of an object's function is under the control of the verb; it may, after suitable checks, determine that the player's request is not reasonable. ("Your load is too heavy. You will have to leave something behind.") This limits flexibility slightly, but it has the advantage that it localizes the tests for a reasonable state.

Presumably, one of the functions will handle the command and print an appropriate response. Should that not happen, the response "Nothing happens" is printed by default. However, care has been taken to ensure that most input commands produce some reasonable response. Indeed, much of the enjoyment of the game is in being allowed to try ridiculous things, and the surprise of having the game understand them.

The functions described so far are invoked in direct response to what the user typed. Background processes, or "demons," are invoked after each input, regardless of its nature. They allow the program to do things independently of the player's actions.

Currently, there are four demons. The first is the "fighting" demon. The residents of the dungeon are frequently hostile; this demon allows them to assault the player unprovoked, and to keep fighting him even if he ignores them.

Next is the driving process behind the "thief," described as a "seedy looking gentleman carrying a large bag." The thief's purpose is to make life difficult for the player by absconding with treasures or other randomly selected objects. In many ways he acts like another, rather hostile and powerful, player in the dungeon.

The third demon is used to warn the player of the presence of hostile forces by causing his sword (if he has it) to glow if there are enemies nearby. It looks at the player's vicinity and prints an appropriate message if the "state of alert" changes; since the thief moves on his own, it is not sufficient to look for hostiles when the player moves.

Last is the "clock" demon. It is the mechanism by which the concept of future time is introduced into the game; arbitrary events can be scheduled for arbitary future times. For example, the lamp can burn out after being on for some number of moves, and wounds inflicted in a fight will eventually heal.

Out of consideration for poor typists, the clock does not tick after unparsed input.

The History of Zork

The existence of Zork is a direct consequence of the existence of two excellent games: Dungeons and Dragons, a fantasy simulation game (not computer based) invented by Dave Arneson and Gary Gygax, and Adventure, a computerized fantasy simulation game originally written by Wil Crowther and later extensively expanded by Don Woods.

Adventure itself was inspired by D&D (as it is familiarly known), in particular a D&D variation then being played out at Bolt, Beranek, and Newman, a Cambridge, Massachusetts, computer firm. It eventually was released to the public, and it became one of the most popular computer games in recent memory.

One laboratory that acquired a copy of Adventure was MIT's Laboratory for Computer Science, with which the designers of Zork (the authors and Bruce K. Daniels) were all then affiliated. In the process of "solving" Adventure, however, the game's deficiencies and the competitive spirit that often animates computer researchers kindled the desire of the authors to write a successor game.

Our natural choice of language was MDL, which is one of the languages in use at LCS. MDL recommended itself for other reasons, however. It is a descendent of LISP and is functionally extensible. It also permits user-defined data types, which is important in a game of "rooms," "objects," "verbs," and "actors." Finally, MDL makes it easy to imbed implicit functional invocations in data structures to tailor the game as described above. The initial version of the game was designed and implemented in about two weeks.

The first version of Zork appeared in June 1977. Interestingly enough, it was never "announced" or "installed" for use, and

the name was chosen because it was a widely used nonsense word, like "foobar."

The original version of the game was much smaller, both geographically and in its capabilities. Various new sections have prompted corresponding expansions in the amount of the universe simulated. For example, the need to navigate a newly added river prompted the invention of vehicles (specifically, a boat). Similarly, the addition of a robot prompted the invention of other actors than the player himself: beings that could affect their surroundings, and so on. Fighting was added to provide a little more randomness in a fairly deterministic game.

The Future of Computer Fantasy Simulation Games

Zork itself has nearly reached the practical limit of size imposed by MDL and the PDP-10's address space. Thus the game is unlikely to expand (much?) further. However, the substrate of the game (the data types, parser, and basic verbs) is sufficiently independent that it would not be too difficult to use it as the basis for a CFS language.

There are several ways in which future computerized fantasy simulation games could evolve. The most obvious is just to write new puzzles in the same substrate as the old games. Some of the additions to Zork were exactly this, in that they required little or no expansion of the simulation universe. A sufficiently imaginative person or persons could probably do this indefinitely.

Another similar direction would be to change the milieu of the game. Zork, Adventure, and Haunt (the CFS games known to the authors) all flow back to D&D and the literary tradition of fantasy exemplified by J.R. R. Tolkien, Robert E. Howard, and Fritz Leiber. There are, however, other milieus; science fiction is one that comes to mind quickly, but there are undoubtedly others.

A slightly different approach to the future would be to expand the simulation universe portrayed in the game. For example, in Zork the concept of "wearing something" is absent: with it there could be magic rings, helmets, boots, etc. Additionally, the player's body itself might be added. For example, a player could be wounded in his sword arm, reducing his fighting effectiveness, or in his leg, reducing his ability to travel.

The preceding are essentially trivial expansions to the game. A more interesting one might be the introduction of magic spells. To give some idea of the kinds of problems new concepts introduce to the game, consider this brief summary of problems that would have to be faced: If magic exists, how do players learn spells? How are they invoked? Do they come in different strengths? If so, how does a player qualify for a stronger version of a spell than he has? What will spells be used for (are they like the magic words in Adventure, for example)? How does a player retain his magic abilities over several sessions of a game?

As can be seen, what at first seems to be a fairly straightforward addition to a game that already has magical elements raises many questions. One of the lessons learned from Zork, in fact, is one that should be well known to all in the computing field: "There is no such thing as a small change!"

A still more ambitious direction for future CFS games is that of multiple-player games. The simplest possible such game introduces major problems, even ignoring the mechanism used to accomplish communication or sharing. For example, there are impressive problems related to the various aspects of simultaneity and synchronization. How do players communicate with each other? How do they coordinate actions, such as attacking some enemy in concert?

Putting aside implementation problems, a multiple-player game would need to have (we believe) fundamentally different types of problems to be interesting. If the game were

cooperative (as are most D&D scenarios), then problems requiring several players' aid in solving them would need to be devised. If the game were competitive and like the current Zork, the first player to acquire the (only) correct tool for a job would have an enormous advantage, to give just one example. Other issues are raised by the statistic that the average player takes weeks and many distinct sessions to finish the game; what happens to him during the time he is not playing and others are?

We believe there is a great future for this type of game, both for the players and for the implementers and designers of more complex, more sophisticated, and—in short—more real simulation games.

Zork Distribution

Zork object code is available from two sources. Complete Zork source listings are not distributed. The MDL substrate of the game, including the parser, data-type definitions, and so on (not the specific dungeon implemented) are available. Write to: P. David Lebling, Room 205, 545 Technology Square, Cambridge, MA 02139. Versions exist for the ITS, Tenex, and Tops-20 operating systems of the DEC PDP-10. To obtain one of these versions or the MDL "substrate" sources, you must enclose a magnetic tape and return postage, specify the operating system on which the program will be run, and what tape formats you can handle. They can make 9-track tapes at 800 or 1600 bpi, using the Tops-20 DUMPER program.

Executable object code of a version of Zork translated from MDL into FORTRAN is available to members of Decus, the Digital Equipment Computer Users Society, One Iron Way, Marlboro, MA 01752. Versions exist for most PDP-11 and VAX operating systems. Order numbers are 11-370B (for RT-11), 11-370C (for RSX11M), or 11-370D (for IAS/VMS).

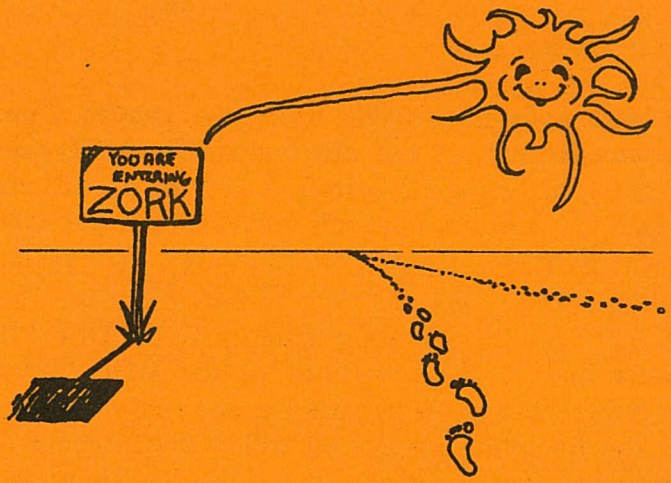
The *MDL Primer and Manual* is available from the MIT Laboratory for Computer Science, Publications, 545 Technology Square, Cambridge, MA 02139. Write for a catalog and price list of LCS publications. □

Bibliography

S.W. Galley and Greg Pfister, *MDL Primer and Manual*, MIT Laboratory for Computer Science, 1977.

P. David Lebling, *The MDL Programming Environment*, MIT Laboratory for Computer Science, 1979.

Gary Gygax and Dave Arneson, "Dungeons and Dragons," TSR Hobbies, Inc., Lake Geneva, WI.



The authors of this article have a variety of interests and skills in addition to their fascination with Zork. P. David Lebling is a staff member of MIT Laboratory for Computer Science; Marc S. Blank is a medical student at the Albert Einstein College of Medicine; Timothy A. Anderson is a member of the research staff at Computer Corporation of America, Cambridge, MA.

Happy Zorking!!