tanulmányok

61/1977

MTA Számitástechnikai és Automatizálási Kutató Intézet Budapest

MAGYAR TUDOMÁNYOS AKADÉMIA
SZÁMITÁSTECHNIKAI ÉS AUTOMATIZÁLÁSI KUTATÓ INTÉZETE

PASCAL.P PORTÁBILIS COMPILER
IMPLEMENTÁLÁSI UTMUTATÓ

Irta:
LEHEL CSABA
BME Folyamatszabályozási Tanszék
ALMÁSI LÁSZLÓ és *LEHEL JENŐ*
MTA SZTAKI Software Osztály

Tanulmányok 61/1977.

A kiadásért felelős:
DR VÁMOS TIBOR

ISBN 963 311 037 8

Készült:
az ORSZÁGOS MŰSZAKI KÖNYVTÁR ÉS DOKUMENTÁCIÓS KÖZPONT
házi sokszorosítójában
F. v.: Janoch Gyula

TARTALOMJEGYZÉK

A tanulmány célja utmutatót adni a portibilis PASCAL.P compiler implementáláshoz. Az utóbbi években megindult a PASCAL nyelv széleskörü elterjedése Európában és az USA-ban egyaránt. Ez a gyors terjedés nagymértékben a PASCAL.P portibilis compilernek volt köszönhető. A szocialista országok közül - a rendelkezésre álló információk alapján - Csehszlovákiában és Lengyelországban foglalkoznak a PASCAL fejlesztésével, illetve alkalmazásával. A tanulmányban közöljük az első hazai PASCAL.P compiler elkészitése során nyert tapasztalatainkat, melyek hozzájárulnak a PASCAL további terjesztéséhez. A tanulmány egyben dokumentáció a PASCAL fejlesztésében rendelkezésre álló software eszközökről.

1. A PASCAL ÉS A PASCAL.P

1.1. A PASCAL nyelvről

A PASCAL programozási nyelvet prof. Niklaus Wirth definiálta 1971-ben, az ETH-n, Zürichben [1]. Az eredeti definició felülvizsgált, módositott változatát - melyet Standard PASCAL-nak is neveznek - 1974-ben tették közzé [2]. Az első PASCAL compiler CDC Cyber 70 tipusu számitógépre készült. E compiler lépésenkénti tökéletesitésének "melléktermékeként" született meg a PASCAL.P portábilis compiler, mely jelen munka kiindulópontját képezte.

A PASCAL nyelv részletes, pontos leirása az irodalomban megtalálható, az alábbiakban csak a nyelv kifejlesztésének célját és sajátos vonásait ismertetjük.

A PASCAL magasszintü, eljárás-orientált, általános célu, ALGOL-szerü programozási nyelv. Kifejlesztésének célja többrétü. Általámosan megfogalmazva a PASCAL nyelv a szisztematikus, strukturált programozás alapkövetelményeit kivánja kielégiteni [4]. Fentiek illusztrálására ragadjuk ki a nyelv néhány fő vonását

- gazdag adatstrukturálási lehetőség, mellyel a programiró a standard adattipusok mellet - saját, a megoldandó feladathoz illeszkedő adattipusokat hozhat létre, ezzel a nyelv eszközeit a különböző feladatokhoz rugalmasan alakithatja.
- a goto nélküli programozás elősegitése, amelyet a nyelv többféle feltételes, illetve ciklus utasitással biztosit /ld. if , case , for , while és repeat utasitások/

- pointerek használata összetett adatstrukturák definiálására és a new standard eljárás változók futási időben történő dinamikus allokálására /generálására/. stb.

A PASCAL nyelv egyik legfőbb erényének azt tekinthetjük, hogy általános célkitüzéseit egyszerü, áttekinthető nyelvi elemekkel és strukturákkal oldja meg, biztositva a magasszintü kifejezésmód és a gépi reprezentációs igények összhangját. A már emlitett pointer tipus mellett ezt a célt szolgálja a halmaz /set/ tipus és az adat tömörités lehetősége /packed strukturák/ is. A halmaz tipus reprezentációi általában a bitenkénti információábrázolást alkalmazzák, ami lehetővé teszi, hogy a halmaz müveleteket egyszerü logikai utasitásokkal valósitsuk meg.
A tömöritett adatstrukturák alkalmazásával egy program memóriaigénye - általában a futási idő növekedése árán - csökkenthető.

A PASCAL nyelv fő alkalmazási területe egyelőre nem a numerikus problémák megoldása, hanem a software fejlesztés /pl. compiler irás/, a programozás oktatása, a software publikálás és dokumentálás.
Megitélésünk szerint, azonban egy adott hardware és software környezetbe gondosan beillesztett PASCAL compiler teljesen kielégitené akár a FORTRAN felhasználók igényeit is.

P é l d á k

1. Tipus definiciók:

```
type
card = array 1..80 of char;
inputfile = file of card;
```

```
bit = boolean;
byte = packed array  0..7  of bit;

letters = set of 'a'..'z';

length = 1..8;
ident  = packed array  length  of char;
symbol = (label,mnemonic);
addrange = 0..32767; oprange = 0..63;
idrec = record
          idname: ident;
          case idtyp: symbol of
               label: (labval: addrange);
               mnemonic:(mnval:oprange)
          end;

hétvezér = (álmos,elől,ond,kond,tass,huba,töhötöm);
osztályzat = (elégtelen,elégséges,közepes,jó,
               jeles);
programnyelv = (algol,fortran,pl/1,pascal);
```

2. Egy program:

```
program példa.1 (output);
type
   tiszt = (alhadnagy,hadnagy,főhadnagy,százados);
var
   rang: tiszt;
   stráf,csillag:integer;

begin
   stráf:=0;

   case rang of
      alhadnagy: begin csillag: =1; stráf:= 1 end;
      hadnagy  : csillag:=1;
```

```
        főhadnagy : csillag:=2;
        százados :  csillag:=3

    end;
end.
```

3. Változó dinamikus allokálása:

```
var  counter: addrange;
     lptr:idptr;
new (lptr,label);
with lptr↑do
    begin idname:='labelnam';
          idtyp:=label;
          labval:=counter
    end;
```

M e g j e g y z é s : a 3. példában használt típusok definiciója az 1. példában található.

1.2. Portabilitás

A programok és programrendszerek portabilitásának kérdése a software fejlesztés fontos, élő problémája. Programok egy számitógépről más számitógépre történő átvitelekor az anyagépen müködő programot a kiválasztott célgépen tesszük müködővé. Egy programot /programrendszert/ akkor tekinthetünk portábilisnek, ha átvitele nem igényel a teljes átirással összemérhető munkát.

A felhasználói programok portabilitása az általánosan használt, szabványos, magasszintü programozási nyelvek alkalmazásával egyszerüen biztositható. Ilyenkor az eredményes átvitel feltétele a használt programnyelv compilerének létezése a célgépen. Programok portabilitásának

biztositása szempontjából tehát alapvető jelentőségü a compilerek gyors terjeszthetősége. Igy egy magasszintü nyelv portábilis compilere egyszerre biztositja a felhasználói programok és a nyelv gyors átvitelét.
Az alábbiakban portábilis compilerekkel és implementálásukkal kapcsolatos néhány gyakorlati követelményt vizsgálunk.

Szokásos megoldás, hogy egy nyelv portábilis compilerét magán a nyelven irják meg ugy, hogy az egy fiktiv kódra fordit, amely utóbbi különböző számitógépeken egyszerüen implementálható. A nyelvnek ilyenkor alkalmasnak kell lenni compiler irásra a generált fiktiv kóddal szemben támasztott követelmény pedig az, hogy legyen szoros kapcsolatban a nyelvvel, ugyanakkor elemi gépi müveleteket irjon le.

Az átvitel gyakorlati megvalósitásához figyelembe kell venni a célgépen adódó memória és futási idő igényt is. Különösen nagy igények esetén az átvitel lehetetlenné, kedvezőbb esetben csak gazdaságtalanná válik.
A portábilis compilerek általánosságuk miatt kisebb hatásfokuak, mint azok a compilerek, melyeket egy adott gép sajátosságainak figyelembevételével irnak.
A továbbiakban röviden leirjuk a PASCAL portábilis változatának a PASCAL.P-nek jellemzőit, implementálási lehetőségeit, majd részletesebben a CDC 3300 gépre implementált PASCAL.P megvalósitásának tapasztalatait, és továbbvitelének módját.

1.3. A PASCAL.P jellemzői

Mivel a PASCAL nyelv compilerek irására előnyösen alkalmazható, a portábilis PASCAL compiler megirható volt ugyanezen a nyelven.

A PASCAL portábilis compilere a PASCAL.P, amely a Standard PASCAL egy részhalmazán iródott és pontosan ugyanezen a részhalmazon irt programokat képes leforditani.

A PASCAL.P korlátozásai a Standard PASCAL-hoz képest:

- file struktura hiánya,
- packed struktura hiánya,
- ugrás tiltása eljárástörzsön kivülre,
- eljárás vagy függvény paramétere nem lehet eljárás vagy függvény.

A korlátozások negy részét portabilitási szempontok indokolják. A PASCAL.P-ben négy előre deklarált textfile-on lehet input, illetve output müveleteket végezni a put és get standard PASCAL eljárások mellet a speciálisan textfile-ok céljára bevezetett read, readln, write, writeln ugyancsak standard PASCAL eljárásokkal.

Ez a négy file:

```
type textfile=file of char;
var  input, prd:textfile (*csak inputra*);
     output, prr:textfile (*csak outputra*);
```

A read, readln, write, writeln eljárások paraméter-listájának formátuma a megszokottól eltérően nem kötött: (f,v1,...vn) ahol f a textfile változó azonositója, vi /i=1...n/ pedig char, integer vagy integer-subrange tipusu változók azonositója.

A PASCAL.P-ben a standard PASCAL dispose eljárást a mark és release eljárások helyettesitik, melyekkel a dinamikus allokálásra használt memória terület nem használt részeit lehet megjelölni, illetve felszabaditani.

A PASCAL.P a forráspforgramokat egy közbülső nyelvre forditja /erre a fiktiv kód, SC-kód, P-kód elnevezések használatosak/.
Müködése egymenetes, azaz a forrásprogramot egyszer olvassa végig és olvasás, értelmezés közben azonnal generálja a fiktiv kódot.

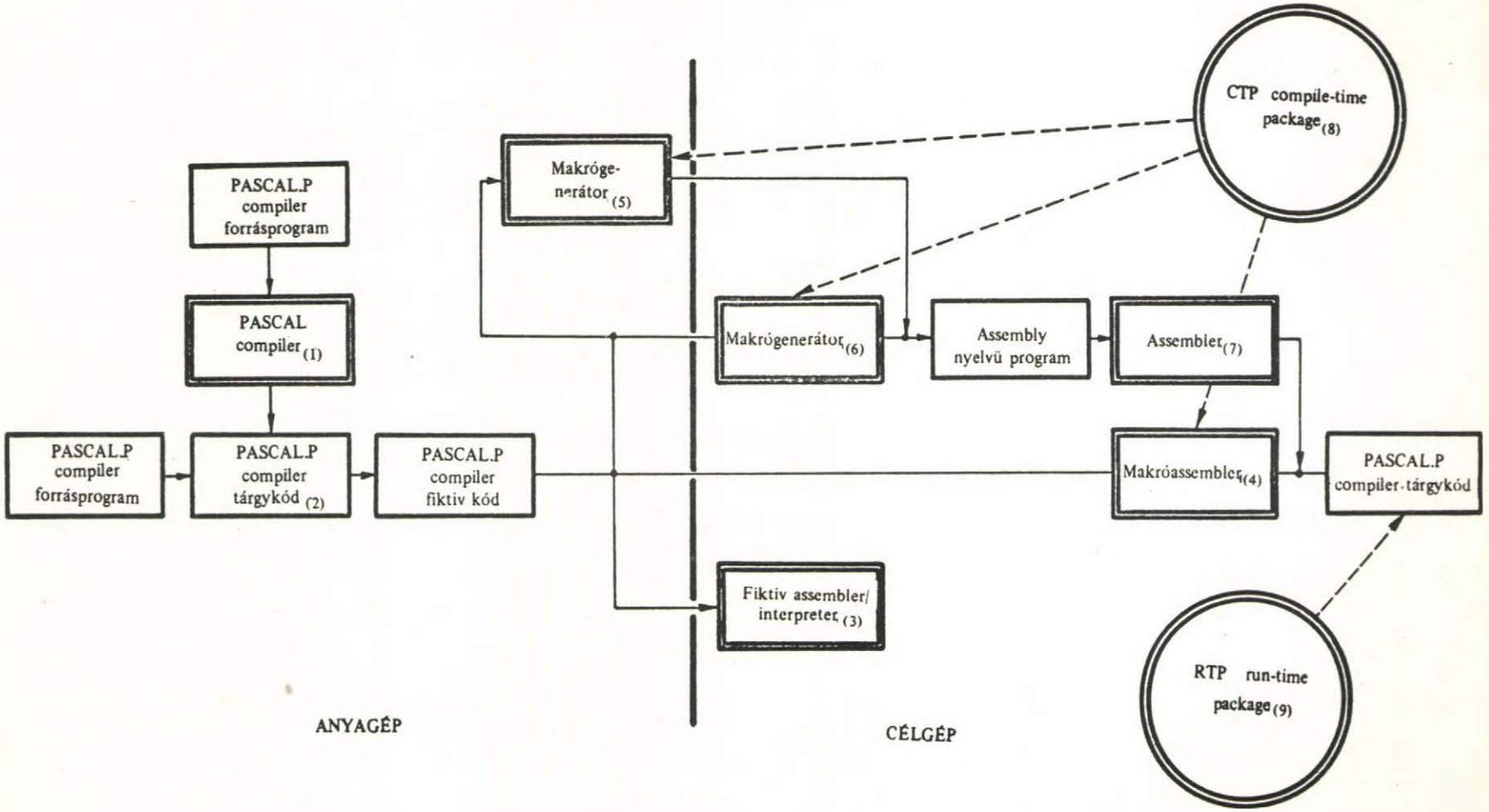
A fiktiv kód implementálásával müködésbe hozott PASCAL.P compiler adott célgépen egyedi módszerekkel jó hatásfoku, a standard PASCAL-t megközelitő compilerré fejleszthető.

1.4. Implementálási lehetőségek

Egy portábilis compiler implementálására különböző stratégiák választhatók az implementáció célja, az anyagép és célgép hardware /software lehetőségeinek, valamint az implementálni kivánt portábilis rendszer jellemzőinek figyelembevételével.

Az implementáció célja alapján két esetet különböztetünk meg:

- interpretálás, amelynek közvetlen célja a portábilis compiler müködésbe hozása a célgépen, eltekintve az igy kapott compiler hatásfokától.
- teljes megvalósitás, amellyel felhasználói szempontból is müködőképes, jó hatásfoku és továbbfejleszthető compilert kivánunk létrehozni.



1. ábra

A PASCAL.P implementálási lehetőségei

A PASCAL.P compiler implementálási lehetőségei az 1. ábrán láthatók. Az ábrán elkülönitve találjuk az anyagépet és a célgépet.

Egy implementációs munka kiinduló lépéseként az anyagép müködő PASCAL compilerével (1) leforditjuk a PASCAL.P PASCAL nyelvü forrásprogramját. A forditás eredménye az anyagépen müködő PASCAL.P compilert ad (2). Ha ezzel leforditjuk az eredeti PASCAL.P forrásprogramot, eredményül fiktiv kódban kapjuk meg a PASCAL.P-t. Amennyiben az implementálás célja interpretálás, a fenti müveleteket egy adott PASCAL.P verzióra elegendő egyszer elvégezni.
A fiktiv assembly nyelven rendelkezésre álló compiler müködtetéséhez a célgép egy már létező nyelvén /pl. FORTRAN-ban vagy a gép assembly nyelvén/ meg kell irni egy fiktiv assembler/ interpretert (3) amellyel a fiktiv utasitásokat a célgépen értelmezzük és végrehajtjuk.
A fiktiv assembler/ interpreter elkészitéséhez szükséges munka a compiler teljes megirásának munkaigényéhez képest csekély. Az interpretálással ugyan müködőképessé hozható a célgépen a PASCAL.P, de az ehhez szükséges memóriaigény és gépi idő igény olyan nagy, hogy ezt az eljárást inkább csak elméleti jelentőségünek tekinthetjük.
A fiktiv assembler/interpreter PASCAL 6000-3.4 nyelven is rendelkezésre áll, de ennek a programnak elsősorban mint dokumentációnak van jelentősége.

Ha a PASCAL.P teljes megvalósitása a célunk, akkor a célgépen egy olyan fiktiv assemblert kell elkészitenünk, mely a gépen végrehajtható gépikódot állit elő a fiktiv nyelv utasitásaiból.

A gyakorlat azt mutatja, hogy a célszerü implementálási stratégiák az interpretálás és a teljes megvalósitás technikák keverékei, azaz a fiktiv kódban irt program müködésbe hozását futási időben, illetve forditási időben felhasznált eszközök is segitik.

A fiktiv kódu programok müködésbe hozásának ezt a módját, másszóval a fiktiv gép megvalósitását, az anyagép és célgép software eszközei támogatják.

Ha a célgép makróassemblerrel (4) rendelkezik, a fiktiv kód utasitásait a makroassembler szabályai szerint definiált makrók hivásainak tekintjük. Ebből a célból a PASCAL.P kódgenerálását módositani lehet az anyagép compilerének felhasználásával.

A célgép makróassemblerével az anyagépen generált fiktiv kód közvetlenül /esetleg egyszerü kódkonverzió után/ lefordítható. Ha a célgép makróassemblerrel nem rendelkezik, a makrókifejtés általános célu makrógenerátorral is elvégezhető, akár az anyagépen (5) akár a célgépen (6). Ekkor a PASCAL.P kódgenerálását a makrógenerátor igényeihez kell illeszteni. A makrógenerátor segitségével a fiktiv kód a célgép assembly nyelvü programjára fordítható.

Mind a makrógenerátor, mind a makróassembler számára definiálni kell a fiktiv assembly utasitások makró-prototipusait /Compile- Time Package, CTP,(8)/. Az assembler (7) vagy makróassembler (4) által generált tárgykód müködtetéséhez szubrutin-csomagot kell irni /Run-Time Package, RTP (9)/.

A PASCAL.P portábilis compiler CDC 3300-ra történő implementálásánál a teljes megvalósitást tüztük ki célul. Az anyagép a svájci EPF* Lausanne CDC Cyber 70 tipusu számitógépe, a SCØPE 3.4 operációs rendszerrel, a célgép pedig az MTA SZTAKI CDC 3300-as számitógépe volt. A megvalósitás nagymértékben kihasználta mind az anyagép PASCAL 6000-3.4 compilere mind a célgép MASTER 4.0 operációs rendszere és CØMPASS makróassemblere által nyujtott lehetőségeket.

Ecole Polytechnique Federal

2. FIKTIV GÉP MEGVALÓSITÁSÁRA ÉPITŐ PASCAL.P IMPLEMENTÁLÁSOK

A PASCAL.P compiler egy fiktiv gépre generál kódot. Ez a gép egy leegyszerüsitett Stack Computer /SC/.

Az SC utasitáskészletének kialakitásával Wirth és Jensen eredetileg nem általános portabilitási követelményeknek kivántak eleget tenni, céljuk a PASCAL compiler lépésenkénti tökéletesitése során a kódgeneráláshoz és cimkijelöléshez jól használható segédlet bevezetése volt [5].

Az SC megvalósitására épitő PASCAL.P implementálás egyrészt megőrzi a termék portabilitását, másrészt kiindulópontja egy hatékony, Standard PASCAL változat helyi kifejlesztésének, amire gyakorlati példák többek között a DEClo,Univac 1100, Xerox Sigma 6 gépek compiler-jei. Ebben a fejezetben először ismertetjük a PASCAL.P/CDC 3300-hoz tartozó SC-t, mely - néhány kisebb változtatástól eltekintve - lényegében azonos az eredetivel. Ezután bemutatjuk a PASCAL.P-nek azt a változatát, mely a célgép akkumulátor regisztereinek tökéletesebb kihasználását teszi lehetővé és igy futási időben és memóriában optimálisabb tárgykód generálását eredményezi egy virtuális-stack computerre /VSC/.

A két implementálási lehetőség eszközeit, összehasonlitó adatait valamint az SC, illetve VSC utasitáskészletét az F1. és F2. Függelék tartalmazza.

2.1 A Stack Computer

Az SC egy véges hosszuságu /elvileg nem korlátozott/ memóriával és három regiszterrel rendelkezik. A memória, mely tárolási egységek /szavak/ cim szerint hivatkozható sorozata két logikai részből áll. Az egyik utasitások tárolására szolgál /utasitás-memória/, a másik rész az adatokat tartalmazza /adatmemória/.
Egy SC utasitás három mezőből áll:

op = utasitás-kód mező,
p,q = utasitás-kódtól függően szignifikáns operandusok /cim illetve paraméter mezők/.

Az SC gondoskodik az utasitás-memóriába helyezett utasitások megfelelő sorrendben történő végrehajtásáról.

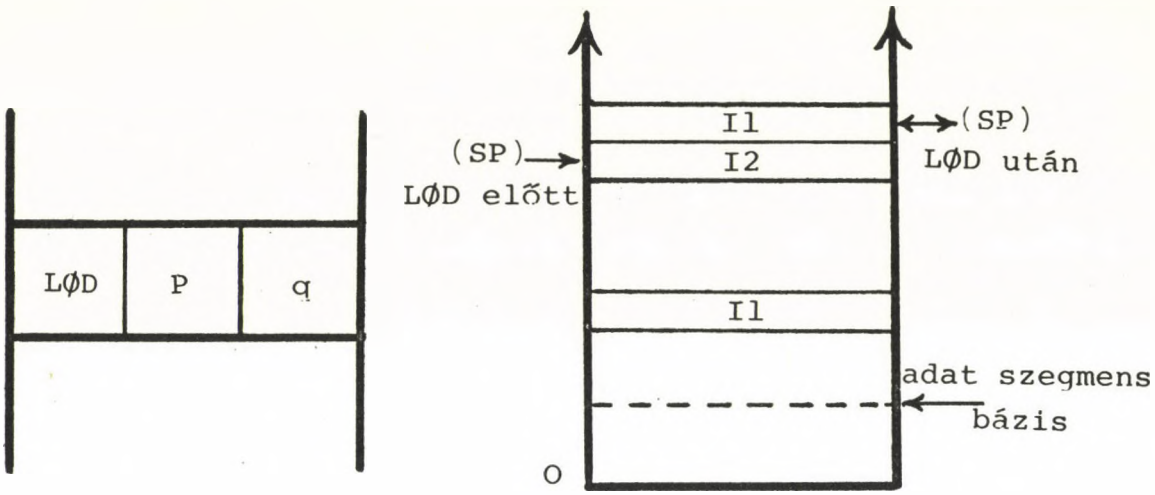
Az utasitások nagy része /pontos definiciókat az F2. Függelék ad/ az SC stack-jére vonatkozó müvelet /stack utasitás/. A stack az adat-memória O cimétől induló, egy SC program futása alatt dinamikusan változó hosszuságu memória rész, melynek tetejére az SP /stack pointer/ regiszterbeli adat-memória cim mutat. Egy-egy stack utasitás végrehajtásakor a stack teteje, illetve az SP regiszter tartalma automatikusan módosul.

P é l d á k

a) Adat betöltés a stack-be

op = LØD
p = az Il integert tartalmazó stack-beli adatszegmens bázis relativ hivatkozási szintje /az indirekt cimzés mélysége/
q = az Il integer relativ cime az adatszegmens bázisára /post-indexelés/.

A betöltés előtt SP tartalma egy egységgel növekszik.



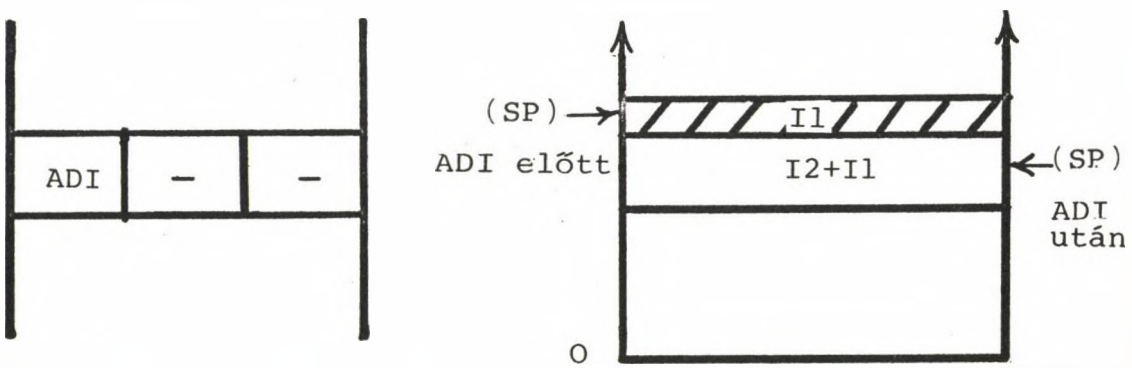
utasitás-memória adat-memória

b) Egész összeadás

op = ADI

p,q= nem szignifikáns

A stack fenti helyzetében kiadott ADI a művelet után SP tartalma eggyel csökken.

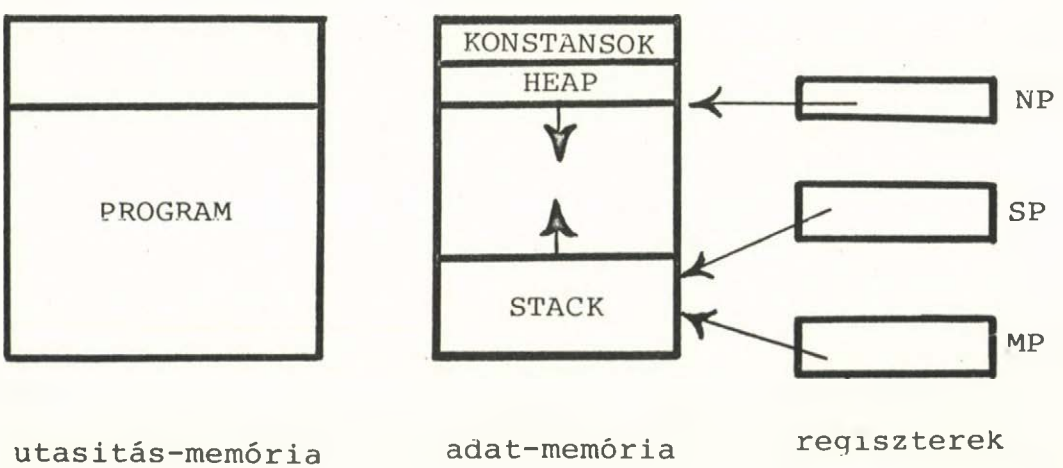


utasitás-memória adat-memória

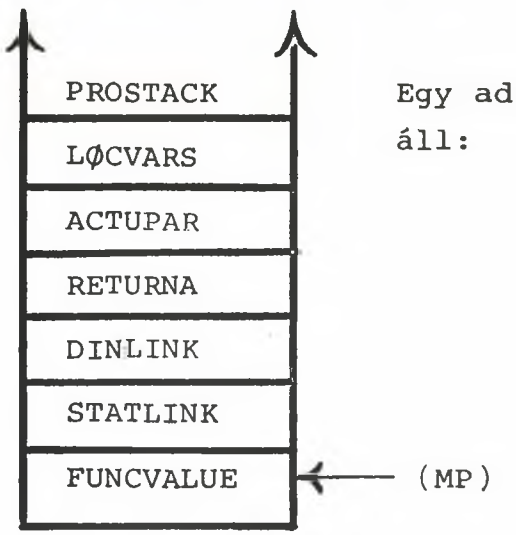
PASCAL eredetü program a Stack Computeren

Az SC-re a PASCAL.P compiler "irja" a programokat, amely munka megkönnyitésére az SC rendelkezik eljáráshivó, adatszegmens kijelölő, összetett elágaztató, visszatéritő utasitásokkal, valamint standard függvényekkel és eljárásokkal, amelyek matematikai függvényeket számitanak /SIN, CØS, LØG stb/, illetve karakter, string, egész változók elemi input /output müveleteit végzik, dinamikus memória kijelölést hajtanak végre stb.

Tekintsük egy PASCAL eredetü program futását a fiktiv gépen. Az utasitás-memória tartalmazza a teljes programot. Az adat-memóriában rögzitett helyet foglalnak el a program végrehajtásához szükséges konstans adatok. Az adat-memória második mezője a Heap elnevezésü csökkenő stack /mely a PASCAL programból dinamikusan allokálható, tehát futási időben kijelölt változókat tartalmaz/. Az NP /new pointer/ tartalma a Heap csucsának cime.



A harmadik adatmező, a Stack, adatszegmensek sorozata. A PASCAL eredetü program futásakor minden aktiv eljáráshoz /procedure, function/ egy-egy adatszegmens tartozik. A felépitett adatszegmens sorozat a program eljárásainak dinamikus hivási sorrendjét tükrözi. Az MP /mark stack pointer/ regiszter tartalmazza az utolsóként hivott eljárás adatszegmensének kezdőcimét /bázisát/.



Egy adatszegmens az alábbi hét mezőből áll:

FUNCVALUE - a function eljárásnál visszaadott érték mezője /function visszatérésekor a hivó procedure adatszegmensében a PROCSTACK mező teteje/

STATLINK - statikus link, pointer az eljárást deklaráló külső eljárás adatszegmensének elejére.

DINLINK - dinamikus link pointer a hivó eljárás adatszegmensére.

RETURNA - az utasitás-memóriában a hivó eljárásba mutató visszatérési cim

ACTUPAR - a paraméterek átvételére szolgáló mező

LØCVARS - az eljárás lokális változóinak mezője

PROCSTACK - az eljárás adatstack-je.

Az adatszegmensben a PROCSTACK mező kivételével minden mező hossza rögzitett. A FUNCVALUE, STATLINK, DINLINK, RETURNA mezők hosszát az SC reprezentálásakor határozzuk meg. Az ACTUPAR és LOCVARS mezők hosszát az ide tartozó paraméterek és változók gépi reprezentációja szabja meg. Ezeket a mezőket futás alatt az eljárás-hivással és adatszegmens betöltéssel kapcsolatos utasitások jelölik ki.

A Stack Computer utasitáscsoportjai, adattipusai

Az SC utasitásai és standard eljárásai funkciójuk szerint az alábbi csoportokba oszthatók:

- Load, Store utasitások
- Integer aritmetikai utasitások
- Logikai utasitások
- Összehasonlitó /relációs/ utasitások
- Set utasitások
- Vezérlésátadó utasitások
- Eljárás hivó, visszatéritő, adatszegmens kijelölő és betöltő utasitások
- Text-file input/output eljárások
- Heap kezelő eljárások
- Matematikai függvények

Az adat-memóriában elhelyezkedő adat tipusa lehet:

- cim
- integer
- character
- boolean
- set

Az SC utasitásainak egy része univerzális több tipusra /pl.load, store/. Az utasitások másik része speciális tipusokon végez müveletet /pl. set utasitások, integer aritmatika stb./

Az utasitások operandusa /p és q/ lehet:

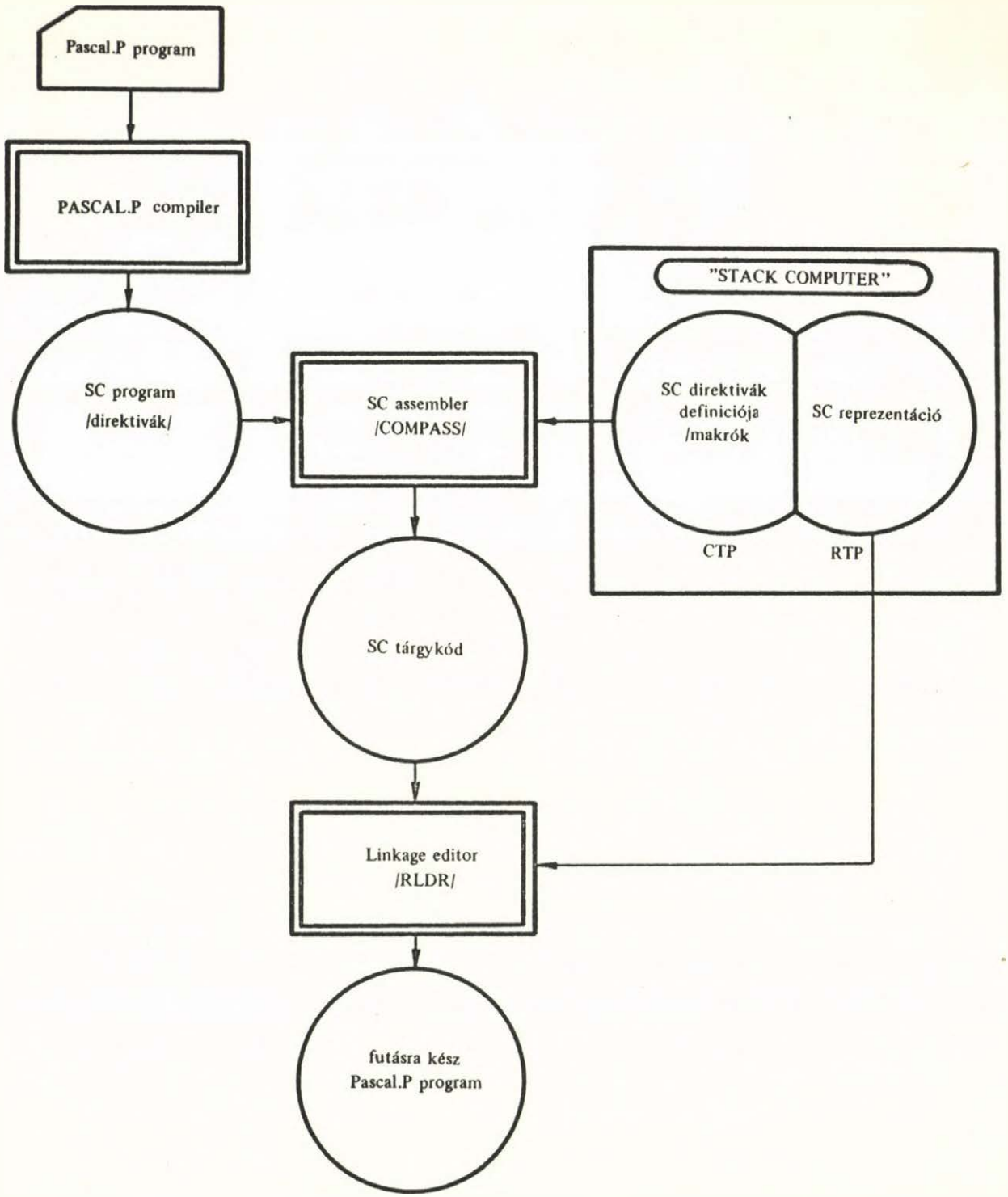
- direkt operandus /konstans/
  a fenti adattipusok bármelyikéből,
- cimhivatkozás az adat-memóriára, vagy utasitás-
  -memóriára.

Az SC kódban operandusként cimke, vagy konstans szerepében szimbolikus nevek is használhatók, továbbá lehetőség van comment sorok alkalmazására.

A Stack Computer megvalósitása CDC 3300 gépen

A CDC 3300 PASCAL.P implementációnál az SC reprezentációját egy szubrutin gyüjtemény /run-time package/ adja, ezt a továbbiakban RTP röviditéssel jelöljük. Az SC assemblerének szerepét a CDC 3300 gép makróassemblerre, a COMPASS tölti be.
A PASCAL.P forrásnyelvü programokat a PASCAL.P compiler leforditja az SC assemblerének szóló direktivákká /COMPASS makróhivások, pseudo-utasitások, comment sorok/. Az SC direktivákat makrók definiálják olymódon, hogy a kapott makróparaméterek alapján előkészitik az SC reprezentációnak szóló operandusokat és az SC utasitás végrehajtását /RTP szubrutin hivás/. A definiációs makrókat és az SC assembler müködéséhez szükséges egyéb kódot a továbbiakban CTP /compile-time package/ névvel jelöljük.

Az SC assembler /COMPASS/ adja azt a tárgykódot, melyet a standard linkage aditor /relocatable loader: RLDR/ összekapcsol az SC reprezentációjával, az RTP-vel. Ennek eredménye a futásra kész PASCAL.P program /2. ábra/



2. ábra

Az SC megvalósitása CDC 3300 számitógépen.

Az SC megvalósitásakor az adattipusok gépi reprezentációjának meghatározása mellett legjelentősebb lépés a CTP és RTP között a funkciók célszerü elosztása.
Ha a CTP definiciós makrói /adott SC program esetén/ teljes müveletek kódjának generálását vállalják, az RTP-ben megvalósitandó müveletek csökkenése mellett a tárgykód memória igényének növekedésével kell számolnunk. Ha a makrók csak az operandusok átadását és a megfelelő művelet szubrutinjának hivását végzik, akkor viszont a tárgyprogram futási ideje növekszik.

Tekintve, hogy a PASCAL.P compiler maga is egy SC program, a CTP és RTP közötti munkamegosztásáról hozott döntés befolyásolja a compiler sebességét, illetve memóriaigényét. A PASCAL.P compilerről a 3. függelék mérése eredményeket közöl, mely segitséget nyujthat a runtime package és a makrók tervezéséhez.

Megjegyzés: A CDC 3300 gép szóhossza 24 bit, az utasitások cimmezeje 15 bit, a belső karakter ábrázolás 6 bites BCD,

regiszterei: A 24 bites akkumulátor regiszter
Q 24 bites regiszter
B1, B2, B3 15 bites cimregiszter.

A programok két, 30K illetve 32K szó hosszuságu memóriarészt használhatnak /1. és 2. chapter/.

A system input és output file-ok formája blokkolt, a blokk-méret 1280 karakter. Egy blokk kártyaképek, illetve sornyomtató sorok karakterláncának fejjel /header/ ellátott rekordjaiból áll.

A felsorolt tulajdonságokból kiindulva az SC CDC 3300-as reprezentációja a következőképpen lett megválasztva.

Utasitás-memória: az 1. chapter /max. 30 K szó/,
ebből 1K helyet foglal el sz RTP

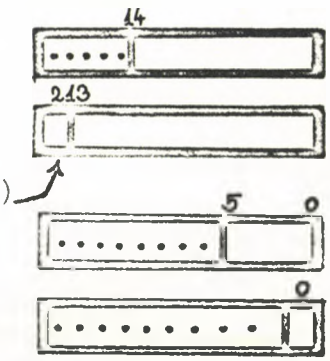
Adat-memória: a 2. chapter /max. 32 K szó/, a program konstans mezőjéből 1K-t foglal az RTP munkaterülete /táblák, bufferek stb./

Az SC regiszterei közül:

SP stack pointer regiszter = B1 indexregiszter
MP mark stack regiszter = B2 "
NP new pointer regiszter = egy kijelölt memória szó az adatmemóriában.

Adat-tipusok reprezentálása:

- cim 1 szóban 15 bit
- integer 1 szóban 24 bit
  (komplemens kódban előjel bit)
- character 1 szóban 6 bit
- boolean 1 szóban 1 bit
- set 2 szóban a halmaz $0 \leq k \leq 47$ elemszámának megfelelő k darab bit



Utasitás operandus akár direkt, akár cimhivatkozás az utasitás-memória egy gépi szavában, vagy A,Q,B3 regiszterekben max. 15 bit.

Az eljárások adatszegmensének mezői:

FUNCVALUE, STATLINK, DINLINK, RETURNA
1-1 szó hosszuságuak

Text-file-ok input/outputját a standard eljárások blokkolva végzik.

Az SC direktivákat definiáló makrók csak operandus átadást és runtime packege szubrutin hivását végeznek /az SC tárgykód memória igénynek optimalizálása érdekében/

2.2 A Virtuális-Stack Computer

A célgép akkumulátor regisztereinek jobb kihasználása, a stack-hivatkozások redukálása, az RTP szubrutinok hivási számának csökkentése a PASCAL.P compiler /azaz a fiktiv kódu programok/ memória igény és futási idő optimalizálása érdekében, voltak azok a szempontok, melyek a Virtuális Stack Computer /VSC/ kidolgozásához vezettek. Bár a PASCAL.P compilernek a VSC-re épülő változata a helyi PASCAL továbbfejlesztésének első állomása, az alkalmazott elv és a megvalósitás megőrizték az előző változatok portabilitását, azaz kiindulási eszközei lehetnek más gépre történő PASCAL.P átvitel munkálatainak is.

A virtuális-stack elve

A virtuális-stack /röviden vstack. lásd [6]/ egy hagyományos memória stack és regiszterek egyesitése. A regiszterek a memória stack logikai folytatását képezik.

Ha a vstack n regisztert használhat, müködését az alábbi PASCAL-ban megadott push és pop müveletekkel lehet definiálni.

```
type  regiszterszám = 0..n;
var   vstacktop, regiszterrészhossza: regiszterszám;

      procedure push;
      begin     if regiszterrészhossza <n
                   then regiszterrészhossza: =
                          regiszterészhossza +1;
```

```
            if vstacktop = n then  vstacktop := 0;
               vstacktop:= succ (vstacktop)
      end (* push *);
   procedure pop;
            if regiszterrészhossza >0
               then regiszterrészhossza: =
                        regiszterrészhossza -1;
            if regiszterrészhossza = 0 then vstacktop:
               then vstacktop: =0;
            if vstacktop >0
               then begin vstacktop: = pred(vstacktop);
                        if  vstacktop = 0 then
                          vstacktop : = n
                     end
end (*pop*);
```

A közölt push és pop PASCAL eljárások a vstack müködtetésének egy lehetséges módját adják meg. Itt a vstack állapotát a felhasznált regiszterek száma /regiszterrészhossza/ és a legfelső regiszter azonositója /vstacktop/ egyértelmüen meghatározzák. Ha éppen nincs regiszterhasználat, akkor /és csak akkor/ vstacktop=0 áll fenn.

Tekintsünk most egy olyan speciális virtuális stack müködést, ahol a vstack csak két regisztert használhat, A és Q regisztereket.

```
type   vstackforma = (s,sa,saq,sqa, sq);
var    vstackállapot: vstackforma;
procedure push;
begin case vstackállapot of
       s  : vstackállapot:=sa;
       sa : vstackállapot:=saq;
       saq: vstackállapot:=sqa;
       sqa: vstackállapot:=saq;
       sq : vstackállapot:=sqa
      end
 end(*push*);
```

```
procedure  pop;
begin  case  vstackállapot of
                s   : vstackállapot: =s;
                sa  : vstackállapot: =s;
                saq : vstackállapot: = sa;
                sqa : vstackállapot: = sq;
                sq  : vstackállapot: = s
       end

end (*pop*);
```

A vstackforma skalárjai egyszerre fejezik ki a használt regiszterek számát, és a legfelső regisztert. /pl. s skalár jelentése: nincs regiszter használat; az sqa jelentése: a vstack memóriarésze felett két regiszter áll, a vstack pedig az A regiszter/.

A virtuális stack computer reprezentációja

A vstack elvnek gyakorlati szempontok miatt /F2. Függelék/ egy egyszerüsitett változatát épitettük be a fiktiv stack computerbe. A kapott VSC kiterjesztése az SC-nek, ezért csak az SC-től különböző vonásait mutatjuk be.

A VSC a három pointer regiszteren kivül /SP,MP,NP/ rendelkezik két munka-regiszterrel A és Q.

A vstack formája lehet: S, SA, SAQ.

A VSC stack-en müveletet végző utasitásainak ismerniük kell a vstack pillanatnyi formáját, a müvelet után pedig meghatározott módon megváltoztatják azt.

Vezessük be a vstack állapotának fogalmát, és jelöljük vss -el /vstack state/.

S állapot ha a vstack S formáju
SA állapot ha a vstack SA formáju
SAQ állapot ha a vstack SAQ formáju
Z állapot ha a vstack S formáju és bármilyen utasitás végrehajtása után is S formáju marad /kényszeritett S állapot/.

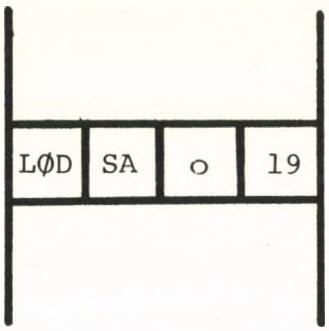
Egy VSC utasitás 4 mezőből áll:

op = utasitás kód mező
vss = virtuális stack állapot mező
p,q = utasitás kódtól függően szignifikáns operandusok.

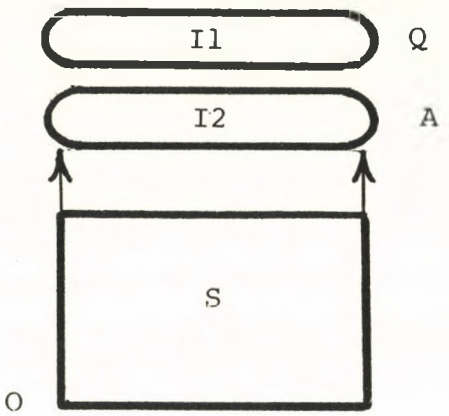
P é l d á k :

a) Il integer betöltése a vstack-be

op = LØD
vss = SA
p = 0 adatszegmens bázis hivatkozási szintje /tehát Il a jelenlegi adatszegmensben lokális változó/
q = 19 az Il relativ cime



utasitás-memória

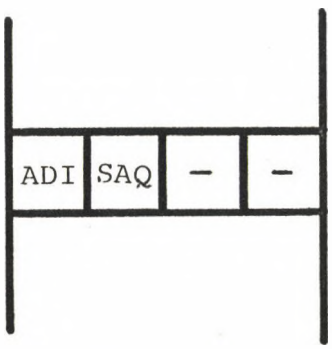


vstack

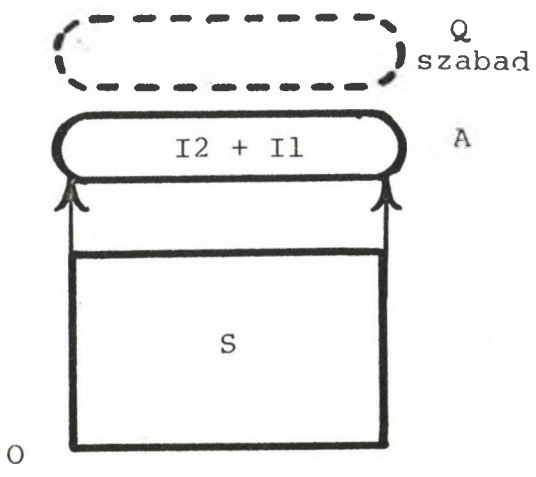
A müvelet végrehajtása után (Q)= Il és a vstack állapota SAQ.

b) Egész összeadás

op = ADI
vss = SAQ
P,q = none



utasitás-memória



vstack

A müvelet után vstack állapota SA /a Q regiszter felszabadul/.

Megjegyzések

A. A PASCAL.P implementálása kapcsán megvalósitott VSC virtuális stack-jének müködését az alábbi függvény-eljárások definiálják:

```
type     vstate = (z,s,sa,saq);
function push (currstate: vstate): vstate;
begin  case currstate of
            z:  push:=z;
            s:  push:=sa;
       sa,saq:  push:=saq
       end
end(*push*);

function pop (currstate:vstate):vstate;
begin case  currstate of
                z: pop:=z;
            s, sa: pop:=s;
              saq: pop:=sa
    end
end (* pop *);
```

B. A VSC programokat generáló PASCAL.P compiler változatban minden VSC utasitáshoz adott a vstack állapot-átmeneti függvénye is, melynek alapján a compiler nyilvántartja a vstack aktuális állapotát.
Igy minden VSC utasitás a megfelelő vss paraméterrel generálódik.

C. A VSC programokban, ha egy utasitáshoz több ágról is eljuthatunk, gondoskodni kell a vstack állapotok egyeztetéséről. Mivel a VSC ugró utasitásai mind S /vagy kényszeritett S/ állapotba viszik a vstack-et, továbbá

a VSC assemblerben csak cimkés utasitásra történhet vezérlésátadás, elég azt biztositani, hogy a cimkés utasitások vss paramétere S /vagy Z/ legyen.

D. A Z vstack állapot /kényszeritett S/ bevezetése a VSC-ben azért szükséges, mert bizonyos adatoknak a program futása során feltétlenül a memóriába kell kerülniük. Ez a helyzet eljárás hivás esetén az aktuális paraméterek átvételekor.

A paraméter átvétel az MST /mark stack/ utasitással kezdődik, a paraméterek kiszámitását, áttöltését stack utasitások végzik, ami a CUP /call user procedure/ utasitással zárul.

Az MST végrehajtása után a vstack Z állapotba kerül, mig a CUP utasitással a vstack Z állapotból S állapotba jut.

E. Az előző megjegyzések illusztrálására mutatjuk be a trans nevü eljárást. Ennek mintájára egyszerüen elkészithető a PASCAL.P programban az az eljárás, mely forditási időben nyomon követi a vstack állapot változását, amit a generált VSC utasitások majd futáskor előidéznek.

/A példa az A.-ban definiált push, pop függvényeket használja/.

```
type  vstate = (z,s,sa,saq);
néhányutasitás  = (adi,lod,eql  (*equality*),
                   mst,cup);
var   currstate: vstate;
      currutasitás:  néhányutasitás;

procedure   trans;
begin case  currutasitás of
```

```
            adi : currstate := pop (currstate);
            lod : currstate := push(currstate);
            eql : if currstate ≠ z then currstate:=sa;
            mst : currstate : = z;
            cup : currstate : = s
            end
        end (*transx*);
```

F. A VSC reprezentálását megkönnyitette a VSC assemblerének használt COMPASS assembler feltételes kódgenerálási lehetősége. A VSC direktivák makró-definiciójában a vss paramétertől függően teljes müveleteket lehet megadni kisszámu gépi utasitással. A run-time package megnövekedése az SC-éhez képest mindössze 1/2 K szó lett, a compiler csökkenése viszont kb. 15K szó /forditási időben a nyereség 20% felett volt/.

3. A PASCAL.P IMPLEMENTÁLÁSÁNAK NÉHÁNY GYAKORLATI KÉRDÉSE

Ebben a fejezetben azokat a problémákat ismertetjük, amelyek a PASCAL.P/CDC 3300 verzió kifejlesztése során merültek fel, és amelyek a későbbi implementálók érdeklődésére számot tarthatnak. A fejezet végén összefoglaljuk egy implementáció lépéseinek célszerü sorrendjét.

A CDC 3300-as implementáció sajátosságát jelentette, hogy az anyagép és a célgép nem állt egyidőben rendelkezésre. A PASCAL.P kódgenerálását ugy kellett az anyagépen elvégezni, hogy közben nem volt lehetőség a célgépen való tesztelésre. A módositott SC kódot mágnesszalagon vittük át az anyagépről a célgépre. A két számitógép eltérő kódkészlete miatt a mágneszalag kódját a célgépen konvertálni kellett. Néhány kisebb hiba miatt az eredeti SC kód nem volt azonnal müködőképes a célgép assemblerével történt lefordítás után. A fenti hibákat az SC kódban kézzel javitottuk ki, de a PASCAL.P müködésbe helyezése ztán a javitásokat PASCAL szinten is elvégeztük, a PASCAL.P forrásprogramban.

Az alábbiakban a munka során felmerült általános jellegü problémákat, illetve azok választott megoldását ismertetjük.

A. Adattipusok reprezentálása

Feladat: A standard PASCAL tipusainak belső ábrázolásához szükséges gépi egységek /bit, byte, szó, duplaszó/ megválasztása

Megoldási kritériumok:

a) A PASCAL.P jelen változatának lefordításához legalább 48 elemü halmaz szükséges

b) A karakterábrázolásánál szempont az ord, illetve char standard függvényeljárások egyszerü megvalósithatósága

c) Az integer tipus esetére biztositandó a numerikus alkalmazásoknál szükséges nagyságrend, a real tipusnál pedig a pontosság
d) összhang alakitandó ki az adattipusok gazdaságos tárolása és a rajtuk végzett elemi müveletek gépidő igénye között

Választott megoldás

Az integer, karakter, boolean és pointer tipus 24 bites gépi szón, a halmazt 48 bites duplaszón ábrázoljuk. A real tipus jelenleg nincs megvalósitva.

B. Az SC-kód konstansainak kezelése

Feladat: Az SC-kódban található adatkonstansok minimális memóriaigényü tárolása

Választott megoldás:

A gépi utasitások cimmezőjében elhelyezhető konstansokat /boolean, char, nil, pointer, 2^{15}-nél kisebb abszolut értékü integer/ direkt operandusnak tekintjük.

A direkt operandusnak nem vehető konstansok közül az integert és halmazt literálként kezeljük. A string konstansokat a fiktiv gép adatmemóriájának konstans mezőjén tároljuk.

Megjegyzés: a nil pointert a CDC 3300 legnagyobb memóriacimenek értékére 77777_8-ra választottuk. A fenti cim adattárolásra nem használható.

C. Az SC-kód karakterfüggése

A PASCAL.P által generált SC-kódban találhatók karakterkód értéktől függő részek.

Feladat: Az anyagép és célgép karakterkódjának különbözősége esetén az alábbi PASCAL.P-ben meglevő karakterkód függésének kiküszöbölése oldandó meg:

- ha karakterkonstanst index vagy intervallum definicióban használunk, az SC-kódban a karakter kódértéke integer konstansként jelenik meg.
- ha karakterkonstansokat case cimkeként tartalmazó strukturát forditunk le, a kapott SC-kódban megjelenő "ugró táblába" a karakterkód értékének megfelelő sorrendben generálódnak az ugró utasitások.

Választott megoldás:

Az SC-kód anyagépen történő generálásakor a PASCAL.P forrásszövegben az index, intervallum és case cimke karakterkonstansait ideiglenesen olyan karakterekkel cseréltük fel, amelyeknek az anyagépen vett belső kódja megegyezett az eredeti karakter célgépen vett belső kódjával.

D. Az SC-kód optimalizálása

Az SC-kódot vizsgálva a kódgenerálás gépiessége gyakran ismétlődő, tipikus utasitássorozatok formájában tünik ki. A PASCAL.P a p=q relációt például az alábbi SC-kódra forditja:

```
lod    (p)
lod    (q)
eql
fjp    (lxxx)
```

A fenti négy SC direktiva egyetlen fiktiv utasitássá történő összevonása mind a futási idő, mind pedig memóriaigény szempontjából kedvezőbb megvalósitást biztositana.

Feladat: az SC-kódban gyakran ismétlődő utasitáscsoportok összevonása egyetlen fiktiv utasitássá.

Választott megoldás:

A PASCAL.P SC-kódjában 13 különféle utasitáscsoport összevonását végeztük el az anyagépen, egy PASCAL.P nyelven irt programmal. Miután a célgépen sikerült müködésbe helyezni a PASCAL.P-t, az optimalizáló program - módositási ciklusok során nyert SC-kódhoz - a célgépen is rendelkezésre állt.

Megjegyzések

A./ Az optimalizálás nélküli SC-kód minimális utasitás-memória igénye meghaladta a CDC 3300-on rendelkezésre álló 30 K szót, igy az optimalizálás a megvalósitás előfeltétele volt.

B./ A VSC-kód az SC-kódhoz hasonlóan optimalizálható.

E. Az SC/VSC programok nyitó és záró blokkjának generálása

A nyitó és záróblokkok egy SC/VSC program elején, illetve végén azokat a szövegeket tartalmazzák, melyl̈el a PASCAL.P által generált fiktiv kódot még ki kell egésziteni a célgépen használt assembler formai elöirásainak megfelelően.

Esetünkben a nyitó blokk a CØMPASS programoknál szükséges IDENT kártyát, a makró definiciós törzseket, external/entry definiciókat tartalmaz. A záróblokk a belépési UJP utasitástól kezdődik a befejező FINIS kártyáig tartó néhány utasitásból áll, melyek már a program assemblálás utáni futtatásához szükségesek.

A nyitó és záróblokkok generálását végezheti a PASCAL.P vagy az RTP. A CDC 3300 változatnál a nyitóblokkot az RTP, a záróblokkot a PASCAL.P generálja.

F. Textfile-ok reprezentálása

Mivel a compiler gyakran végez char input/output müveletet egy fordités során, nyilvánvalóan a textfile reprezentációnak és a read, readln, write, writeln müveletek megoldásának jelentős hatása lesz az implementált compiler idöigényére.

A választott reprezentációnkban ezért illeszkedtünk az operációs rendszer standard blokkolásu file szerkezetéhez, de kiküszöböltük a blokkos átvitelből adódó felesleges adatátmozgatásokat.

Textfile-okon az input/output megkönnyitésére bevezetett fogalom a standard PASCAL-ban a "line marker", mellyel a textfile karaktersorozata sorokká tördelhető.

A "line marker" beállitását az outputra használt file-on egyedül a writeln standard eljárás végzi. Az input file-on az eoln PASCAL függvénnyel lehet a line markert vizsgálni.

Feladat: a "line marker" reprezentálása

Megoldási lehetőségek

a) speciális line marker karakter, vagy karakterkombináció,
b) rögzitett sorhosszuság,
c) a sorokhoz külön-külön kiegészitő információként hozzávett hosszuság.

Mi ez utóbbit választottuk. A writeln eljárás az aktuális sor elejére /header-ébe/ helyezi a sor hosszának szavakban vett értékét /ha kell, blank-karakterekkel kiegésziti előbb a sort/. Az f input file-on az eoln akkor ad true értéket, ha egy sor utolsó karakterére kiadott get(f) után ujabb get(f) lett kiadva. Az ezután kiadott get(f) abban az esetben, hogyha van még sor a file-on törli az end of line feltételt és f↑-be helyezi a következő sor első karakterét. Ellenkező esetben az end of line feltétel mellett bekövetkezik az"end of file" feltétel is.

Az end of line feltétellel egyidőben f↑ = balnk áll fenn, de egy következő karakter read nem ezt a blanket, hanem az uj sor első karakterét fogja beolvasni. Az input file-on tehát nem soronkénti,hanem folyamatos feldolgozásra is lehetőség van. Megjegyezzük, hogy standard input, vagy prd file-okat használó PASCAL.P progaramokban a program elinditása előtt az RTP behelyezi a file első karakterét a buffer változóba /f↑-ba/.

G. A compiler változóinak inicializálása "vágással".

A PASCAL.P compilernek több eljárása azzal foglalkozik, hogy a compilerben definiált strukturákat táblákat kiépitse, kezdeti értékeket adjon meg.

Egy compiler müködésében ezek a feladatok minden egyes fordítás alkalmával ugyanugy lesznek végrehajtva, és az eredmény is ugyanaz a memória felépités, kitöltés. Az inicializáló eljárások az inicializálás után is bent maradnak a memóriában /a compiler részeként/ a használt konstansok a fordítás alatt végig lekötik az adatmemória konstans mezőjét.

Ennek megszüntetésére alkalmaztuk az SC/VSC "vágásának" módszerét, melynek elve röviden a következő. A C compilert két részre vágjuk, A /inicializáló/, B /fordító/. Az A felépiti a C stack-jét, heap-jét, és az adat-memóriát egy bináris file-ra viszi. A továbbiakban C helyett elég a B-t használni, csak B inditása előtt a lináris file-ról be kell tölteni az adat-memóriáját.

H. Fordítási és futtatási opciók

Feladat: A PASCAL.P compilernek hogyan lehet megadni, hogy a forrásprogramot tartalmazó kártyaképekről csak $k \leq 80$ karaktert értelmezzen /tehát pl. egy javitórendszer által a kártyakép végére helyezett sorszámokat ne a forrásprogramhoz tartozó karakterekként kezelje/.

Egy megoldás

A compilerben az adatmemória egy meghatározott cimére globális változót deklarálunk. Ennek a fordítás megindítása előtt az RTP adja meg azt a k kezdeti ér-

téket, amit a felhasználó a compilert meghivó vezérlőkártya paraméterei között előirt.

Ugyanugy egyéb felhasználói opciók is bevezethetők /kereszt-hivatkozási lista, értékhatár ellenőrzés stb./ a helyi továbbfejlesztés során.

Az implementálás lépései

A PASCAL.P implementálásának a stack computer megvalósitására épitő módszerében igen lényeges szempont a compiler müködő változata és a forrásnyelvü változat közti párhuzam folyamatos fenntartása. Ez módositási ciklusokon keresztül valósitható meg. Egy módositott ciklus olyan PASCAL.P forrásnyelvü programot érintő és /vagy a müködő compiler gépi kódján végzett javitások sorozatából áll, melynek végeredményeképpen a PASCAL.P program leforditása ugyanazt a gépi kódot eredményezi, mint a forditást végző compiler kódja.

A továbbiakban összefoglaljuk az inplementálás előre látható fontosabb lépéseit és a lépésekhez szükséges software feltételeket.

1. A PASCAL.P alapvető adattipusai reprezentálásának meghatározása a célgépre, és a PASCAL.P gépfüggő konstans-definicióinak megváltoztatása.
2. A stack computer reprezentációjának /run time package/ megtervezése, és az esetleg felmerülő PASCAL.P forrásnyelvi szintü változtatások kijelölése /pl. VSC esetén/.
3. A PASCAL.P program 1. és 2. szerinti javitása, valamint a kódgeneráló rész módositása abból a célból, hogy formailag is a célgép makró assembleréhez igazodó fiktiv kódot kapjuk. A PASCAL.P program forrásnyelvi szintü javitásainak elvégzéséhez az anyagépen természetesen PASCAL vagy

PASCAL.P compiler szükséges.

4. Az anyagépen elkészitett PASCAL.P compiler változat és a megfelelő assembler-kód átvitele a célgépre, szükség esetén konvertáló programok alkalmazásával.

5. A CTP és az RTP közti munkamegosztás rögzitése.

6. A CTP elkészitése és az assembler kód leforditása, memória igényének értékelése.
   Ennél a lépésnél felmerülhet a makró assembler bővitésének problémája, valamint az assembler-kód memóriaigényre történő optimalizálási lehetőségeinek áttekintése.

7. Az RTP elkészitése és az assembler-kód belövése.

8. A müködő compiler program illesztése a célgép software-éhez /könyvtározás, hivási mechanizmus, file kezelés/.

9. A PASCAL.P compiler felhasználói számára compiler opciók lehetősége, külső file-ok használati módja, futási hibaüzenetek generálása azok a feladatok, melyeket egy user run-time package-el teljesithetünk.

4. A PASCAL.P LEHETŐSÉGEINEK ÉRTÉKELÉSE

A közeljövőben haszánkban is várható a PASCAL nyelv szélesebb körü terjedése. Ennek alapfeltétele természetesen az, hogy a különböző számitógépeken rövid időn belül hatékony forditóprogramok álljanak rendelkezésre. Ezek kifejlesztéséhez megfelelő eszköz a PASCAL.P.

Anyagépként rendelkezésre áll az MTA CDC 3300-as gépe, ahol a PASCAL.P SC ill. VSC változata generálható.

A PASCAL.P implementálásához becslésünk szerint ötödannyi idő szükséges, mint egy compiler teljes megirásához.
Egy célgépre átvitt PASCAL.P compiler helyei továbbfejlesztésével kiküszöbölhetők a portabilitásból adódó hatékonysági veszteségek, illetve a standard PASCAL nyelven vett korlátozások. Ezek a munkák nem igényelnek több időt, mint a PASCAL.P implementálása.

Átvitel CDC 3300-ról más gépre

Néhány elkezdett, illetve tervezett fejlesztés:

- Megindult a PASCAL.P implementálása az ESSZR gépcsalád R 50 gépére.

- Tervezzük egy run time rendszer megirását R 10 gépére, mellyel anyagépen leforditott PASCAL.P programokat lehet müködtetni.

- Tanulmányozzuk a PASCAL.P compiler kisgépekre /TPA 70, R 10/ történő átvitelének lehetőségeit.

Továbbfejlesztés a CDC 3300 gépen

Amellett, hogy a PASCAL.P továbbvitelében számitógépünk már betöltheti az anyagép szerepét, célszerünek látszik compilerünk többirányu továbbfejlesztése.

A. A hatékonyság növelése érdekében cél makroassembler készitése. Jelenleg a PASCAL.P által generált SC, illetve VSC kódot a CØMPASS assembler forditja. Ennek a második fázisának az időigénye /különösen a VSC esetén alkalmazott sok feltételes makró utasitás miatt/ tul nagy. /Pl. mig az SC, ill. VSC kód generálását,8, illetve 5 perc alatt végzi a compiler a PASCAL.P forrásnyelvi programjáról, addig a fiktiv kód leforditása CØMPASS-szal mintegy 26, ill. 22 perc perc./ A cél makró - assemblerrel végzendő második fázishoz módositani kell az első fázis kódgenerálását ugy, hogy a VSC direktivák, paraméterek mnemonikus generálása helyett bináris utasitáskód és paraméter értékek generálódjanak. Ez mellékhatásként csökkenti az első fázis idejét is.

   A második fázis memória optimalizálást is végezhet a 3. pontban ismertetett elven. Ehhez jelentős támpontul szolgálnak az F4. Függelékben közölt statisztikai adatok.

B. Lehetőség van egy Standard PASCAL compiler fokozatos kialakitására. A real-tipus implementálásához runtime rutinok megirása és a kódgenerálás némi változtatása szükséges. Az 1. fejezetben leirt PASCAL.P korlátozások közül lényegesnek tartjuk még a packed strukturálás és a file tipus megvalósitását.

C. A PASCAL felhasználói szolgáltatásokkal történő bővitése. A határ vizsgálatát /index, range/ és a változó kereszt-hivatkozási lista opcionális beépitését, valamint az eljárások szeparált fordithatóságának bevezetését tartjuk a leglényegesebbnek.

A szerzők köszönetüket fejezik ki Ch. Rapin professzornak és J. Ramanoelina tanársegédnek, az EPF Lausanne oktatóinak, jelen munka támogatásáért és az abban nyujtott segitségükért.

F Ü G G E L É K E K

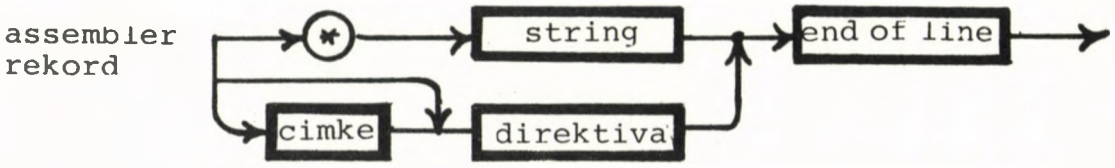
F1. A STACK-COMPUTER ÉS VIRTUÁLIS-STACK COMPUTER DEFINICIÓJA

Minthogy a két fiktiv gép assemblere azonos, utasitáskészletük nagy része közös, a meghatározásokat egyszerre adjuk meg mindkét gépre, a vstack állapot-átmeneteket pedig a függelék végén foglaljuk össze.

Az assembler formai leirása

Az assembler leirására szintaxis diagramokat használunk. A diagram blokkjai közül a kerekitett szélüek terminálo szimbólumok, a többiek jelentését a diagramot követő megjegyzés, illetőleg ujabb diagram tisztázza. Egy diagram blokkjainak tetszőleges nyilfolytonos bejárása /kezdőponttól végpontig/ szintaktikusan helyes egységet definiál.

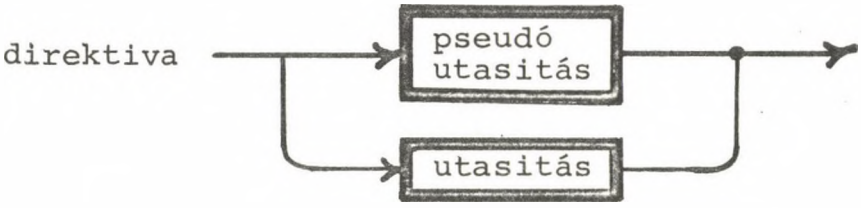
Megjegyzés: a nyitó- és záró blokkok szolgáltatják azt a keretet, melyek a fiktiv gép megvalósitásakor a leforditott programok tényleges futásához szükségesek. A nyitó blokk tartalmazhatja a direktivákat definiáló makró törzseket, cimkék entry ponttá nyilvánitását, external hivatkozások kijelölését stb., a záró blokkba kerülhetnek a program futását elinditó és befejező direktivák.



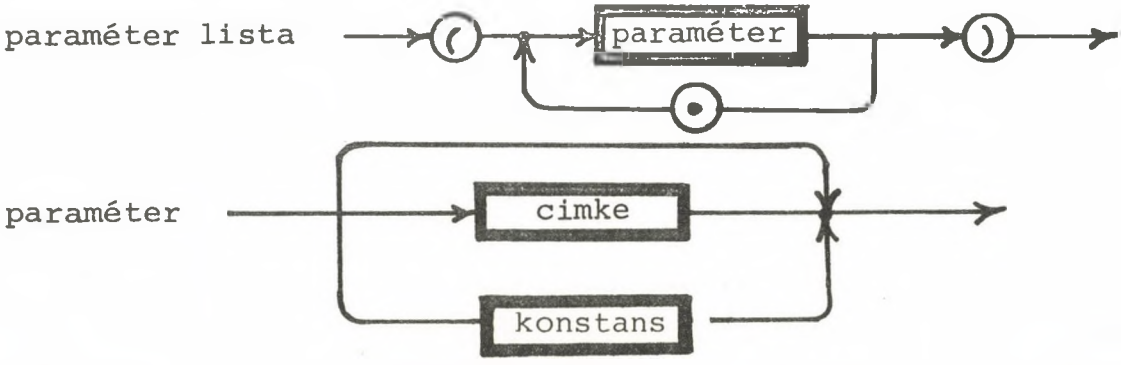
Megjegyzések: az end of line lehet logikai fogalom (pl. assembler rekord hossza = 72 karakter), vagy ténylegesen létező (pl. két ':' karakter jelzi a sor végét);

A '*' karakterrel kezdődő sorok commentek, az assembler program olvashatóságát segitik /a PASCAL.P compiler pl. minden user eljárás törzsének generálása előtt commentként megadja az eljárás nevét/

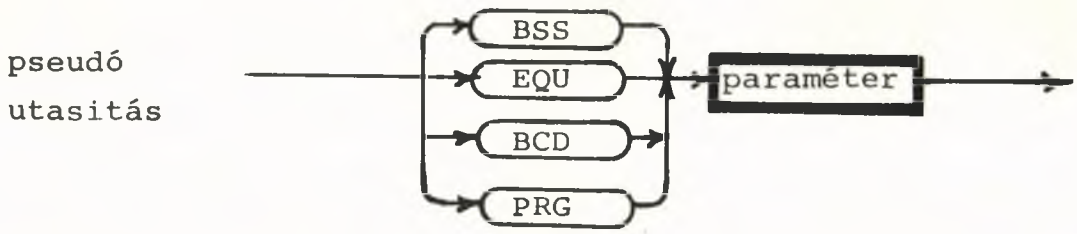
Megjegyzés: az 'L123' jellegü cimkék hossza legfeljebb 8 karakter, ezeket a PASCAL.P cim, vagy integer tipusu konstansok szimbolikus megnevezésére használja,



Megjegyzések: a direktiva mnemonikja mindig a 10. karakter pozicióban kezdődik, a hozzá tartozó paraméterek pedig a 20. karakter poziciójában;
a paraméter formája függ a direktivától.

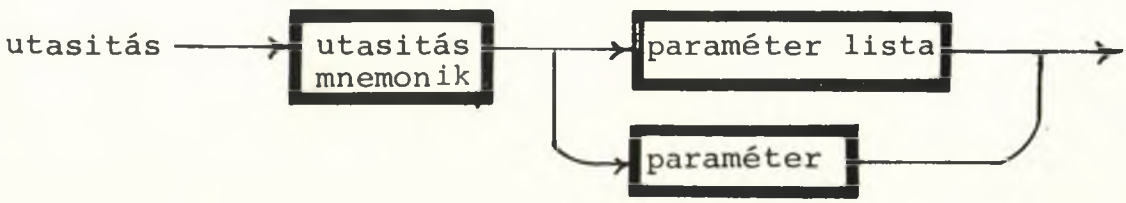


Megjegyzés: a konstans paraméterek formáját, jelentését és a cimke paraméterek jelentését a direktivák részletes leirásával együtt adjuk meg. /Az "üres" paraméter 0 integer értékü/



Megjegyzések:

- a BSS paramétere 0, az assembler a hozzátartozó cimkéhez az utasitásszámláló /location counter/ aktuális értékét rendeli;
- a EQU paramétere integer konstans, ezt az értéket adja az assembler a hozzátartozó cimke szimbolumnak;
- a BCD paramétere 1,000C alaku, ahol C tetszőleges karaktert jelent, amely karakter belső kódját az assembler az SC adat-memóriájának soronkövetkező szavába helyez;
- a PRG-nek nincs paramétere, hatására az assembler az adatmemóriacim fejlesztését befejezve visszatér az utasitás-memóriához.



Megjegyzések:

SC esetén két utasitás az 'UJP' és 'RTJ' után egyetlen cimke szimbólum áll, mint paraméter;

A paraméter lista a 20. karakter pozicióban kezdődik;
A mnemonik 3 vagy 4 karakter az A..Z intervallumból.

Az SC és VSC utasitásai

- Az utasitásokat funkciójuk szerint csoportositva közöljük.
  Bár a PASCAL.P compiler jelenlegi változata 59 darab 3-4 karakteres alap mnemonikot kezel, felsorolásunk csak azokat az utasitásokat tartalmazza, melyek a PASCAL.P implementálásához feltétlenül szükségesek. További /pl. real aritmetikával kapcsolatos/ utasitások értelmezése az implementálókra marad.
- Az utasitások operációs kódjaként a compiler mnemonik-tömbjének indexét használjuk /ami nam azonos az [5]-ben található kóddal/.
- Mivel az assembler az utasitás paraméter listájának elemeit a fiktiv gépi utasitás operandusaivá "alakitja" nem teszünk különbséget az utasitás paraméter és utasitás operandus megnevezések közt.
- A program adatainak 5 tipusát különböztetjük meg
  - cim (vagy pointer)
  - integer
  - character (belső-kód érték)
  - boolean (0 v. 1)
  - set (bitsorozat) formailag integer

  A fiktiv gép bizonyos utasitásai univerzálisak több tipusra /pl. LØD, STR/, mások speciális tipusokon végeznek müveletet /lásd pl. LØDS, LDCN/.

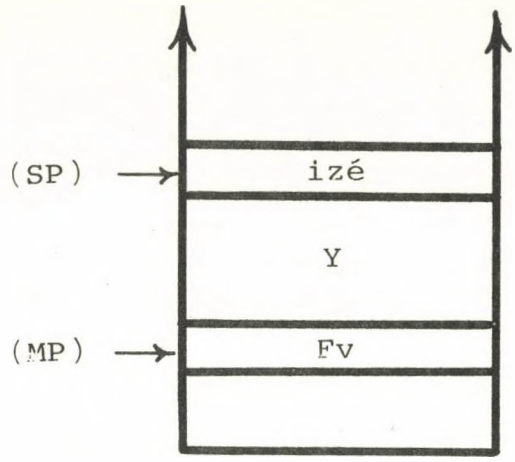
Az utasitások paramétere /p,q/ lehet:

- bármilyen tipusu direkt operandus,
- cimhivatkozás bármilyen tipusu adatra, vagy szimbolikus hivatkozás esetén az utasitásmemóriára
- a legtöbb utasitásban a p integer tipusu direkt operandus szerepe az, hogy megadja az aktuális adatszegmenstől induló és az adatszegmensek STATLINK-jén visszafelé haladó többszörös indirekt hivatkozás lépésszámát, ezt hivatkozási szint operandusnak nevezzük.
- Az eljárások adatszegmensén /stack-jén/ történő változás leirására a következő jelölést vezetjük be

stack változás: [x] → [x']

Itt X és X' a stack formája egy utasitás végrehajtása előtt, illetve után. A stack csucsát mindig a stack forma jobb oldala jelzi /pl. az [S int1 int2] formáju stack csucsán illetve alatta int2 illetve int1 adatok helyezkednek el/. Ha egy utasitás stack változása [S int1 int2] → [S sum], akkor az utasitás nem érinti a stack S-sel jelölt részét, végrehajtásakor az int2 és int1 adatok lekerülnek a stack-ről, a stack csucsa a sum-mal jelölt adatot fogja tartalmazni.

A stack forma baloldala az aktuális eljárás adatszegmensének báziscimétől ábrázolja a stack-et. Pl. az ábrán látható stackhelyzetet [Fv Y izé] formával irhatjuk le.



Load utasitások

LAØ (q) Load base-level address

op = 37
q = cim tipusu direkt operandus
stack változás: [S]→[S,q]

LDØ (q) Load from base-level address

op = 39
q = cim tipusu hivatkozás
legyen x a q cim tartalma
stack változás: [S]→[S×]

LAD (p,q) Load address

op = 50
p = hivatkozási szint operandus /integer,direkt/
q = cim tipusu direkt operandus
q relativ cim a p által meghatározott b bázisra.
Legyen ad = b + q
stack változás: [S]→[S ad]

LAC ( ) Load address of constant

op = 38

A stack computer adat-memóriájában a konstans mező location counterének aktuális értéke legyen clc

stack változás: [S] → [S clc]

LDC (p) Load constant

op = 51

p = nem set tipusu, egyébként tetszőleges direkt operandus

stack változás: [S] → [S p]

Megjegyzés: az LDC utasitásnak különböző változatai léteznek a betöltendő konstans tipusa szerint, amit a stack computer megvalósitásának szempontjai indokolnak.

LDCC /p/ Load character constant
LDCN /p/ Load nil pointer
LDCL /q/ Load long constant

Az előbbi utasitásokban p direkt operandus és rendre character, cim, tipusu. A q olyan "nagy integer" az LDCL utasitásban, amely a direkt operandusok számára felhasználható bit számmal nem reprezentálható, ezért q értékét az assembler elhelyezi a stack computer adat-memóriájának soronkövetkező konstansmező cimére ahonnan majd futáskor cimhivatkozás után kerül q a stackre.

A VSC ezek közül csak az LDCN és LDCL változatokat használja.

LDØS (q) Load set from base-level

op = 20
q = cim tipusu hivatkozás

Legyen set a q cim tartalma

stack változás: [S] → [S set]

LØDS (p,q) Load set from address

op = 24
p = hivatkozási szint operandus /integer, direkt/
q = cim tipusu hivatkozás

Legyen ad = b+q, ahol b a p által meghatározott báziscim, és set az ad cim tartalma,

stack változás [S] → [S set]

LDCS (p,q) Load set constant

op = 27
p,q= set tipusu direkt operandusok

Megjegyzés: a két paraméter lehetőséget ad egy konstans halmaz két diszjunk darabból történő összerakására. Ha a set reprezentációja nagyobb méretü, mint amit direkt operandusként az assembler kezelni tud, akkor a konstans halmazt az assembler az adat-memória konstans mezőjébe helyezi és arra cim hivatkozást generál.

Legyen conset a p és q részhalmazokból konkatenációval kialakitott halmaz

stack változás: [S] → [S conset]

Indexelt load utasitások

IDX (q) Indexed fetch

op = 35
q = integer, direkt operandus
stack változás: [S addr] → [S ×]

ahol × az addr + q cim tartalma

IDXS (q) Indexed fetch set

op = 29
q = integer direkt operandus
stack változás: [S addr] →[S set]

ahol set az addr + q cim tartalma.

Store utasitások

SRØ (q) Store at base-level

op = 43
q = cim, direkt operandus

stack változás: [S ×] → [S]
Az × adat a q cimre kerül.

STR (p,q) Store

op = 56
p = hivatkozási szint operandus /integer, direkt/
q = cim, direkt operandus
stack változás: [S ×] → [S]

az × adat a b + q cimre kerül, ahol b a p alapján képzett báziscim

STØ ( ) Stack store

op = OO
stack változás: [S addr ×] → [S]
× adat az addr cimre kerül

SRØS (q) Store set at base-level

op = 28

q = cim, direkt operandus

stack változás [S set] → [S]

A set adat a q cimre kerül

STRS (p,q) Store set

op = 25

p = hivatkozási szint /integer, direkt/

q = cim, direkt operandus

stack változás: [S set] → [S]

A set adat a b+q cimre kerül /b a p által meghatározott bázis-cim/.

STØS ( ) Stack store set

op = 26

stack változás: [S addr set] → [S]

Összehasonlító utasitások

Az ebbe a csoportba tartozó utasitások a stack-ben közvetlenül vagy cimhivatkozással meghatározott két adat között végeznek el vizsgálatot egy feltétel teljesülésére, és 'true' vagy 'false' boolean értéket adnak vissza a vizsgálat eredményétől függően.

A stack változás mindenütt [S x y] → [S boole]

```
   EQL              Test on' = '
   EQL              Test sets on '='
op = 47
   NEQ              Test on  '≠'
   NEQS             Test sets on '≠'
op = 55
   GEQ              Test on  '≥'
   GEQS             Test sets on '≥'
op = 48
   LEQ              Test on  '≤'
   LEQS             Test sets on '≤'
op = 52
   GRT              Test on  '>'
op = 49
   LES              Test on  '<'
op = 53
```

A további összehasonlitó utasitások az előbbiek változatai. Itt azonban a kiindulási [S×y] stack formában × és y kezdőcimei egy-egy adatsorozatnak, melyeket szavanként hasonlit össze az utasitás.

A q paraméter /integer, direkt operandus/ adja meg az adatsorozatok hosszát.

EQLM	(q)	Test sequences on '='
NEQM	(q)	Test sequences on '≠'
GEQM	(q)	'≥' test az első nem egyenlő elempáron
LEQM	(q)	'≤' test az első nem egyenlő elempáron
GRTM	(q)	'>' test az első nem egyenlő elempáron
LESM	(q)	'<' test az első nem egyenlő elempáron

Logikai utasitások

NØT — Boolean 'not'

op = 19
stack változás: [S b] → [S $\bar{b}$]
ahol $\bar{b}$ a b boolean tipusu adat negáltja

AND — Booelan 'and'

op = 04
stack változás: [S a b] → [S c]
itt c az a és b boolean adatok logikai szorzata: $c = a \wedge b$

IØR — Inclusive 'or'

op = 13
stack változás: [S a b] → [S c]
Az a,b boolean adatok közötti "megengedő vagy" müvelet eredménye a c /mely akkor és csak akkor 'false', ha a is b is 'false'/

Integer aritmetika

Ebben a csoportban minden utasitás stack változása

[S i j] → [S k] (i,j,k egész adatok)

ADI — Addition

op = 02
k = i+j

DVI — Integer Division

op = 06
k = i/j egész része

MØD — Modulus

op = 14
k = i/j egész osztás maradéka

MPI — Multiplication

op = 15
k = i * j

SBI — Subtraction

op = 21
k = i - j

Halmaz müveletek

Ebben a csoportban a stack változás:

[S e f] → [S g] (e,f,g set tipusu adatok)

UNI — Union

op = 45
g = e∪f

INT — Intersection

op = 12
g = e∩f

DIF — Set difference

op = 05
g = e-f — g az e halmaz f-be nem tartozó elemeinek halmaza

Eljárások müködtetésével kapcsolatos utasitások

MST (p) Mark stack

op = 31
p = hivatkozási szint operandus
MST az eljárás hivás előkészitésének első utasitása, feladata a hivandó eljárás adatszegmensének első mezőit lefoglalni illetve kitölteni

stack változás: [H] → [H fv stl dinl]
Az utasitás megfelelő hosszuságu helyet biztosit az esetleges függvény érték számára, ezt jelöli fv;
a dinl cim a hivó eljárás adatszegmensének bázis cime /dinamikus link/;
az stl cim statikus link azon eljárás adatszegmensének bázis cime, amelyben a hivott eljárás definiálva van, azaz amelynek a hivott eljárás lokális eljárása;
a statikus link meghatározását az utasitás a p hivatkozási szint paraméter alapján végzi.

CUP (p,q) Call user procedure

op = 46
p = integer, direkt operandus
q = szimbolikus cimhivatkozás

A CUP utasitás a p paraméter és az SP regiszter tartalma alapján kiszámitja a hivandó eljárás adatszegmensének bázis cimét és a mark pointer /MP/ regiszterbe helyezi MP := (SP)-p-3 , továbbá elhelyezi a RETURNA adatszegmens mezőn az utasitás-memóriába mutató visszatérési cimet /hivási hely + 1/, majd ugrik a q paraméterben megadott cimre.

stack változás: [HM] → [M]

H a hivó eljárásban a hivás megkezdésig (MST végrehajtásig) felépitett stack rész, M jelzi az eljárás-hivás előkészitése során (MST-től a CUP utasitásig) felépitett stack részt.

A hiváskor átadandó aktuális paraméterek az MST és CUP utasitások között stack utasitások segitségével kerülnek az adatszegmensbe. (lásd az 5. függelék példaprogramját)

ENT (q) Enter data segment

op = 32
q = szimbolummal kifejezett integer, direkt operandus.

Minden eljárás ENT utasitás nyit meg, feladata a stack printer /SP/ inicializálása.

stack változás: [M] → [M R]

R egy stack-rész az eljárás lokális változói illetve munka-mezői számára fenntartva. SP uj értékét a q paraméter határozza meg:

SP := (MP) +q-1

RETP Procedure return

op = 08

stack változás [HS] → [H]
H jelöli a hivó stack formáját S pedig a visszatérő eljárásban épitett saját stack-et.
(H kezdőcimét a dinamikus link adja)

RET (p) Function return

op = 42
p = karakter, direkt operandus
stack változás: [H funcval S] → [H funcval]
H a hivó eljárás stackje.
S a function eljárás saját stack-jének a funcval adat utáni része, funcval a function eljárásban kiszámitott függvény érték. A function tipusát a p paraméter jelzi.

Egyéb utasitások

MØV (p) Move data sequence

op = 40
p = integer, direkt operandus
stack változás : [S ad1 ad2] → [S]
p jelenti az ad2 cimtől kezdődő adatsorozat hosszát, mely sorozatot az utasitás áthelyezi ad1 cimtől kezdve.

NGI ( ) Negate integer

op = 17
stack változás: [S i] → [S j] ahol i,j integer adatok, és j = - i

DCR (p) Decrement address

op = 31
p = integer, direkt operandus
stack változás: [S a] → [S b] ahol a,b cimek és b = a-p

INC (p) Increment address

op = 34
p = integer, direkt operandus
stack változás: [S a] → [S b] ahol a,b cimek és b = a+p

lXA (p) Compute indexed address

op = 36
p = integer, direkt operandus

stack változás [S addr ix] → [S iad]
addr, iad cim adatok az ix integer adat
iad = addr + ix.p

Megjegyzés:
az utasitás megadhatja pl. a p hosszu elemekből álló addr cimtől induló tömb ix indexü elemének cimét.

SGS (p) Generate singleton set

op = 23
p = integer, direkt operandus
stack változás [S] → [S set]

A set egy egyelemü-részhalmaz, melyet az elemnek a lineárisan rendezett alaphalmazban elfoglalt pozíciójával határoz meg a p paraméter ($0 \leq p \leq 47$)

INN ( ) Test on set membership

op = 11
stack változás: [S int e] → [S bool]

int integer jelzi az e /lineárisan rendezett/ halmaz egy elemének pozicióját, a bool boolean adat megadja, hogy az igy meghatározott elem benne van-e e halmazban.

FJP (p) False jump

op = 33
q = szimbolikus cimhivatkozás

A q egy kód-memóriába mutató cim, melyre az utasitás csak akkor ugrat, ha a stack csucsán 'false' érték van

stack változás: [S bool] → [S]

Speciális VSC utasitások

BRU (q,vss) Branch unconditional

op = 57
q = szimbolikus cimhivatkozás

A virtuális stack S állapotba kerül, majd feltétel nélküli ugrás a q által meghatározott kód-memória cimre.

CSP (q,vss) Call standard procedure

op = 30
q = szimbolikus cimhivatkozás

A q a VSC egy standard eljárásának a neve. Az eljárásra szubrutin ugratás történik, a visszatérés S vstack állapottal a CSP-t követő utasitásra adja a vezérlést.

PØP (vss) Vstack pop

op = 59

Ez a vstack kezeléshez bevezetett segéd-utasitás a vstack-et S állapotba viszi, azaz az esetleg használt A,Q regisztereket betölti a stack memóriába.

CAS (lmin, lmax, q, vss) Case

op = 58
lmin, lmax = integer direkt operandusok
q = szimbolikus cimhivatkozás

A PASCAL case utasitás támogatására bevezetett segéd-utasitás.
stack változás: [S k] → [S]

Amennyiben lmin≤k≤lmax, ugratás történik a q cimtől kezdve felsorolt BRU utasitások közül a (q + (k - lmin) + 1)-edikre.

Ha k az lmin..lmax intervallumon kivül esik, ugrás a q cimre.

Speciális SC utasitások

UJP q Unconditional jump

stack változás: [S] → [S]
A BRU utasitás SC-változata

RTJ q Return jump

A CSP utasitás SC-változata

q = szimbolikus cimhivatkozás /eljárás neve/
stack változás: [S x] → [S f]

/itt X jelöli a standard eljárás stacken kapott adatait, f pedig a függvénytől visszakapott értéket/

XJP p Indexed jump

q = szimbolikus cimhivatkozás
/A CAS utasitás az XJP-nek összetettebb VSC változata/

stack változás: [S i]→ [S]

Feltétel nélküli ugrás q + i -re.

Standard függvények, eljárások

EØF End of file ellenőrző függvény
ELN End of line ellenőrző függvény

stack változás: [S fb]→[S bool]

fb = megadja a vizsgálandó file bufferének cimét
bool = a vizsgálat eredményét jelző boolean.

PUT Pascal.P. PUT müvelet eljárása
GET Pascal.P. GET müvelet eljárása
RLN Pascal.P. READLN müvelet eljárása
WLN Pascal.P. WRITELN müvelet eljárása
stack változás: [S fb] → [S]
Az fb jelöli ki azt a file-t /illetve bufferét/, melyre a müveletet végre kell hajtani.

RDC Read character
RDI Read integer
stack változás: [S add fb] → [S]
fb = input file /buffer/ kijelölése
add= az a cim ahová a karakter, ill. integer beolvasását végre kell hajtani.

WRC Write character
WRI Write integer
stack változás: [S k ℓ fb] → [S]
fb = output file /buffer/ kijelölése
k = a kiirandó konstans /char., vagy integer/
ℓ = a kiirandó karakterszám

WRS Write character string
stack változás: [S sadd ℓ ls fb] → [S]
fb = output file /buffer/ kijelölése
sadd = a character string kezdő cime
ls = a string tényleges hossza
ℓ = a kiirandó karakterek száma

Megjegyzések:

- A fiktiv gép, akárcsak a PASCAL.P text-file-ok használatát engedi meg, tehát integer input/output müveleteknél karakterkód ~ integer érték konverciákkal adódik a számjegyek stringjének megfelelő egész érték, illetve az értéknek megfelelő string.

- Azokban az output eljárásokban, ahol a kiirandó karakterek száma /1-lel jelölt stack adat/ nagyobb, mint a ténylegesen kiirandó string hossza, balról blank kiegészités történik, ha pedig kisebb, akkor a string csonkitatlanul iródik ki.

NEW PASCAL NEW standard eljárás
stack változás [S add n] → [S]

Az eljárás n hosszuságu memória-darabot foglal le az adatmemória HEAP mezőjéről, azaz n -el csökkenti NP/new pointer/regiszter tartalmát és a kezdő cimet az add cimre teszi.

SAV Save NP /new pointer/ regiszter

stack változás: [S add] → [S]

Az eljárás az add cimre helyezi az NP regiszter tartalmát.

RST Reset NP /new printer/ regiszter
stack változás: [S np] → [S]

Az eljárás betölti NP regiszterbe az np cimet.

Megjegyzés:
a PASCAL standard "dispose" eljárását a PASCAL.P -ben "mark" és "release" standard eljárások helyettesitik, melyeknek fiktiv gépi megfelelői a SAV és RST eljárások.

A virtuális stack állapot-átmenet

- A VSC utasitásokban mindig az utolsó paraméterlista elem a vss vstack állapot operandus.
A vstack állapotai a következő vstack formákkal szemléltethetők:

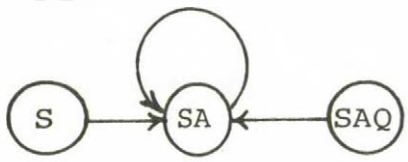
S, SA, SAQ ahol

S = valódi memória stack rész
A,Q = regiszterek

A forma jobb oldala a vstack csucsát mutatja.

Egy utasitás összes vstack változását nevezzük a vstack állapot átmeneti függvényének, és jelölésére gráfot választunk:

Pl:



jelentése:

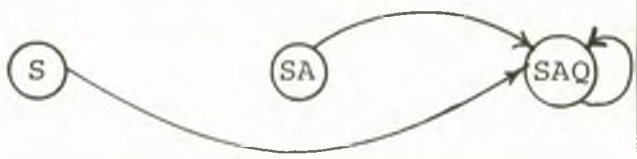
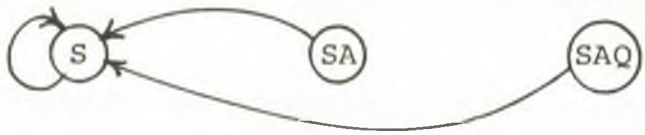
$$\begin{cases} [S] \to [SA] \\ [SA] \to [SA] \\ [SAQ] \to [SA] \end{cases}$$

A negyedik vstack állapot a "kényszeritett S", mely állapotban egyetlen utasitás sem használhat regisztert, azaz a vstack változás ebben az állapotban minden utasitásra $[Z] \to [Z]$, /ahol Z valódi memória stack/.

- A VSC utasitások állapot-átmeneteit olyan gyakorlati szempontok szerint alakitottuk ki, melyek figyelembe vétele biztositotta a célul kitüzött memória-igény, futási idő optimalizálását a PASCAL.P compilerre:

  a) A és Q regiszterek szerepe a CDC 3300 gépen nem szimmetrikus /Q-val kevesebb müvelet végezhető, mint A-val/
  b) A választott adat-reprezentációban mindkét regiszterben minden adattipus tárolható a virtiális stack-en /amit indokol az AQ párral végezhető néhány CDC 3300 utasitás/. Az itt közölt állapot-átmeneteket szemléltetésnek szánjuk, kiegészitésül a PASCAL.P/CDC 3300 compiler ver-

sion 2-ben található vstack-átmeneti táblához.

Utasitások	Vstack állapot - átmeneti gráf
NØT	
AND, IØR, IXA	
NGI, DCR, INC	
ADI, SBI, SRØ, STR	
LAØ, LAC, LDØ, LAD, LDC, LØD, LDCN	
DVI, EØF, INN, IDX, EQL, GEQ, GRT, NEQ, LEQ, LES	
DIF, INT, SGS, MØD, MPI, LDØS, LØDS, LDCS, UNI, IDXS	
STRS, STØS, MØV, SRØS, CSP, ENT, EJP, MST, RET, CUP, BRU	

F2. A PASCAL.P implementálás eszközei

Az itt felsorolt eszközök egy része programok, listák formájában áll rendelkezésre, más részük irodalmi hivatkozás.

a) PASCAL Revised Report, 1974. [2]
b) PASCAL.P/CDC 3300 Felhasználói leirás
c) PASCAL.P/CDC 3300 Compiler forrásnyelvü alakban PASCAL.P-ben irva kb. 3600 80-karakteres kártyakép/
d) PASCAL.P/CDC 3300 compiler fiktiv SC kódban /kb. 15000 40-karakteres kártyakép/
e) A fiktiv Stack Computer és assemblerének definiciója

A fiktiv VSC megvalósitását választó implementációhoz a fenti a) és b) anyagok mellett a következő eszközök használhatók:

c^*/ PASCAL.P/CDC 3300/Version 2/ compiler forrásnyelvü alakban
d^*/ PASCAL.P/CDC 3300/Version 2/ compiler a fiktiv VSC kódban
e^*/ A Virtuális-Stack Computer és assemblerének definiciója

Jelen tanulmányon kivül, mely e/ és e^*/ eszközöket tartalmazza a fiktiv Stack Computerre vonatkozóan használható még:

- PASCAL.P compiler Implementation Notes [5]
- PASCAL 6000-3.4 nyelven irott assembler/interpreter az eredeti Stack Computerre /mérete 1000 80-karakteres kártyakép/. Elsősorban dokumentációs célt szolgál.

F3. A PASCAL.P/CDC 3300 karakterkészlete és kódja

A. A PASCAL.P compilert és SC kódját tartalmazó mágneszalagra irt karakterek kódja.

	0	1	2	3	4	5	6	7	8	9
0	:	1	2	3	4	5	6	7	8	9
10	0	=	≠	<	%	[	BLANK	/	S	T
20	U	V	W	X	Y	Z	]	,	(	↱
30	≡	∧	-	J	K	L	M	N	Ø	P
40	Q	R	V	$	*	↑	↓	>	+	A
50	B	C	D	E	F	G	H	I	<	.
60	)	≥	¬	;						

Megjegyzés: a ↱ karakter a PASCAL.P-ben nem értelmezett

B. A PASCAL.P Standard PASCAL-tól eltérő alternativ szimbólum értelmezése.

jel megnvevezése	Standard PASCAL	PASCAL.P
kettőspont	:	: vagy % vagy ..
aposztróf	'	↓
nem egyenlő	< >	< > vagy ≠
nagyobb egyenlő	> =	>= vagy ≥
kisebb egyenlő	< =	<= vagy ≤
logikai negáció	not	NOT vagy ¬
logikai és	and	AND vagy ∧
logikai vagy	or	ØR vagy ∨
kommentár kezdet, vég	{...}.	(* ... *)

C. A mágnesszalag fizikai jellemzői:

A CDC 3300 gépen a felirás 7 trackes framekkel történik, páros vagy páratlan paritásbittel, 200, 556 vagy 800 BPI süsüréggel. A file mark kódja: 15 (karakter), file vége jel: két file mark.

I. TÁBLÁZAT

MNEMONIC	Z	S	SA	SAZ	SUMMA
FNT	1	89	0	0	== 90
LOD	60	540	170	15	== 785
LOC	495	242	1149	83	== 1969
GRT	0	0	1	29	== 30
FJP	0	29	724	1	== 753
LAC	0	37	226	3	== 266
BCO	1573	0	0	0	== 1573
FRG	0	266	0	0	== 266
LAO	1	358	44	116	== 519
CSP	0	0	33	181	== 214
STR	0	15	476	0	== 491
LOO	187	1011	242	13	== 1453
LEQ	0	0	0	43	== 43
DCR	0	0	3	245	== 248
IXA	1	0	2	382	== 385
IDX	54	0	265	41	== 360
EQL	0	0	0	335	== 340
BRU	0	731	20	0	== 751
BSS	230	0	0	0	== 230
LES	0	0	0	23	== 23
ADI	1	0	15	60	== 76
INC	2	0	306	3	== 311
SRO	0	10	207	0	== 217
RETP	0	86	0	0	== 86
EQU	89	0	0	0	== 89
GEQ	0	0	0	7	== 7
STO	0	2	25	400	== 427
EOF	0	0	1	0	== 1
NOT	0	12	79	2	== 93
MST	0	953	6	0	== 959
CUP	959	0	0	0	== 959
NEQ	0	0	0	192	== 192
AND	0	4	48	2	== 54
ICR	0	4	22	0	== 26
SBI	7	0	0	22	== 29
EQLM	0	0	0	9	== 9
LAD	205	86	13	0	== 304
MPI	0	0	1	9	== 10
CAS	16	0	1	3	== 20
MOV	0	0	7	280	== 287
LOCN	1	56	183	9	== 249
LESM	0	0	0	3	== 3
LODS	88	2	30	4	== 124
INN	0	0	0	75	== 75
LOOS	12	0	19	6	== 37
UNI	90	0	0	24	== 114
STRS	106	0	0	2	== 108
LOCS	97	9	23	18	== 147
NGI	1	0	2	0	== 3
RET	0	3	0	0	== 3
DIF	2	0	0	0	== 2
IDXS	0	0	0	1	== 1
MOD	0	0	0	1	== 1
DVI	0	0	0	2	== 2
SGS	1	0	0	1	== 2
NEQS	0	0	0	1	== 1
STOS	0	0	0	2	== 2
LOCL	0	1	0	0	== 1
SROS	0	0	0	9	== 9
				TOTA	== 15867

II. TÁBLÁZAT

COUNT	INSTRUCTION GROUPS OF LENGTH 3 4 , ENDED BY FJP			
3	LOD	CUP	AND	FJP .1
1	LOD	LOD	CJP	FJP .1
1	IDX	LOD	CUP	FJP .1
1	LOD	LOD	CJP	FJP .1
3	MST	LOD	CUP	FJP .1
1	LOD	IDX	CUP	FJP .1
1	LOD	IDX	CUP	FJP .1
4	LOD	LOD	CUP	FJP .1
3	LOD	LOD	CUP	FJP .1
1	MST	LOD	CUP	FJP .1
3	IDX	LOD	CUP	FJP .1
1	IDX	CUP	NOT	FJP .1
3	LOD	CUP	NOT	FJP .1
1	LOD	CUP	NOT	FJP .1
2	LAO	CSP	NOT	FJP .1
1	LODS	LOCS	NEQS	FJP .2
1	STR	LOD	IDX	FJP .2
1	LOD	IDX	FJP .2	
1	LOCS	UNI	INN	FJP .2
1	LOD	IDXS	INN	FJP .2
3	LOD	LOCS	INN	FJP .2
4	LOD	LOCS	INN	FJP .2
8	LOD	LODS	INN	FJP .2
5	LOC	LODS	INN	FJP .2
1	IDX	LODS	INN	FJP .2
1	CUP	BGS	LOD	FJP .2
1	DVI	STR	LOD	FJP .2
1	IXA	STR	LOD	FJP .2
1	LAD	CUP	LOD	FJP .2
1	LOC	CUP	LOD	FJP .2
17	MST	CUP	LOD	FJP .2
7	LOD	FJP .2		
2	LOD	LAO	LESM	FJP .2
1	LOD	LAD	LESM	FJP .2
2	LOC	GRT	AND	FJP .2
1	INN	NOT	AND	FJP .2
2	LOD	NOT	AND	FJP .2
10	LOD	LOL	AND	FJP .2
4	LOC	EQL	AND	FJP .2
1	IDX	EQL	AND	FJP .2
1	LOD	NEQ	AND	FJP .2
7	LOCN	NEQ	AND	FJP .2
3	LOCS	INN	AND	FJP .2
2	LOC	LEQ	AND	FJP .2
1	LOD	LOD	EQLM	FJP .2
1	LOD	LAD	EQLM	FJP .2
2	LOD	LAO	EQLM	FJP .2
1	IXA	LAO	EQLM	FJP .2
1	LODS	INN	IOR	FJP .2
2	CUP	NOT	IOR	FJP .2
1	LOC	LES	IOR	FJP .2
2	LOD	CUP	IOR	FJP .2
2	LOD	EQL	IOR	FJP .2
6	LOC	EQL	IOR	FJP .2
7	LOC	GRT	IOR	FJP .2
7	LOD	LOD	NEQ	FJP .2
3	LOD	LOD	NEQ	FJP .2
1	IDX	LOD	NEQ	FJP .2
1	LOD	LOD	NEQ	FJP .2
1	LOD	LOD	NEQ	FJP .2
2	IDX	LOD	NEQ	FJP .2

43	LOD	LOCN	NEQ	FJP .2
12	IDX	LOCN	NEQ	FJP .2
44	LOD	LOCN	NEQ	FJP .2
22	IDX	LOC	NEQ	FJP .2
2	LOD	LOC	NEQ	FJP .2
19	LOD	LOC	NEQ	FJP .2
1	EQLM	IDR	NOT	FJP .2
2	LOD	IDX	NOT	FJP .2
2	CUP	LOD	NOT	FJP .2
1	BSS	LOD	NOT	FJP .2
17	STR	LOD	NOT	FJP .2
3	BRU	LOD	NOT	FJP .2
1	LOD	NOT	FJP .2	
12	LDOS	INN	NOT	FJP .2
19	UNI	INN	NOT	FJP .2
9	LDOS	INN	NOT	FJP .2
6	LDOS	INN	NOT	FJP .2
2	SRO	LOD	NOT	FJP .2
2	LOD	NOT	FJP .2	
1	LAO	EOF	NOT	FJP .2
1	LOD	LOD	GEQ	FJP .2
1	LOD	LOD	GEQ	FJP .2
1	LOD	LOD	GEQ	FJP .2
1	IDX	LOC	GEQ	FJP .2
2	LOD	LOC	GEQ	FJP .2
1	LOD	LOC	GEQ	FJP .2
1	LAD	MOV	LOD	FJP .2
1	LOD	CUP	LOD	FJP .2
1	CUP	BSS	LOD	FJP .2
8	EQU	ENT	LOD	FJP .2
1	SRO	BRU	LOD	FJP .2
1	BRU	BRU	LOD	FJP .2
1	CUP	BRU	LOD	FJP .2
1	CSP	BRU	LOD	FJP .2
4	LAO	CSP	LOD	FJP .2
1	LOD	SRO	LOD	FJP .2
2	LOD	FJP .2		
2	LOD	LOD	LES	FJP .2
1	LOD	LOD	LES	FJP .2
1	SBI	LOD	LES	FJP .2
4	LOD	LOC	LES	FJP .2
4	LOD	LOC	LES	FJP .2
1	LOD	LOD	LES	FJP .2
1	LOD	LOD	LES	FJP .2
1	LOD	IDX	EQL	FJP .2
15	LOD	LOD	EQL	FJP .2
5	IDX	LOD	EQL	FJP .2
16	LOD	LOD	EQL	FJP .2
2	LOD	LOCN	EQL	FJP .2
2	IDX	LOCN	EQL	FJP .2
12	LOD	LOCN	EQL	FJP .2
1	MOD	LOC	EQL	FJP .2
27	LOD	LOC	EQL	FJP .2
43	IDX	LOC	EQL	FJP .2
139	LOD	LOC	EQL	FJP .2
2	IDX	LOD	EQL	FJP .2
1	LOD	LOD	EQL	FJP .2
3	LOD	LOD	EQL	FJP .2
1	LOD	LOD	LEQ	FJP .2
9	IDX	LOC	LEQ	FJP .2
4	LOD	LOC	LEQ	FJP .2
8	LOD	LOC	LEQ	FJP .2
1	IDX	LOD	LEQ	FJP .2
16	LOD	LOD	LEQ	FJP .2
1	LOD	LOD	GRT	FJP .2
2	LOD	LOD	GRT	FJP .2
1	IDX	LOD	GRT	FJP .2
1	ADI	LOD	GRT	FJP .2

1	LOD	LOD	GRT	FJP .2
1	LOD	IDX	GRT	FJP .2
1	IDX	LOC	GRT	FJP .2
5	LOD	LOC	GRT	FJP .2
3	LOD	LOC	GRT	FJP .2

TOTAL= 753

III. A PASCAL.P compiler /version 2/ eljárásainak hivási gyakorisága

A compiler 89 eljárásban összesen 278.176 hivás történ a PASCAL.P leforditása közben. A táblázat felsorolja a leggyakoribb eljárásokat, (megadva, hogy 1000 eljáráshivás közül hányszor lettek meghivva)

eljárás neve	hivások száma 1000 hivás közül
NEXTCH	529
INSYMBØL	83
MNGEN	49
VSTACK	49
CØMPTYPES	29
SEARCHID	23
FIELDLIST	23
LØAD	20
GEN2	19
FACTØR	18
TERM	18
SIMPLEXPRESSIØN	17
GEN 1	16

eljárás neve	hivások száma 1000 hivás közül
EXPRESSIØN	14
ENDØFLINE	13
STATEMENT	13
SELECTØR	12

IV. A VSC-t reprezentáló RTP szubrutinjainak hivási gyakorisága

Az ØP fiktiv gépi utasitást vagy eljárást VSS vstack állapotban megvalósitó RTP szubrutint jelöljük ØP.VSS szimbólummal. A táblázat néhány RTP service rutinról is közöl adatokat, ez esetben röviden a service rutin funkcióját adjuk meg.

Az alábbi táblázat csak az érdekesebbnek itélt adatokat tartalmazza.

RTP szubrutinok	hivások száma
ENT.Z, MST.Z, CUP.Z, RETP.Z RET.Z	278.176 268.836 9.340
RDC.SAQ, EØF.SA	148.665
ELN.SA	297.328
WRC.SAQ	210.080
WRS.SAQ	23.440
WRI.SAQ	37.799
WLN.SA	19.604
NEW.SAQ	2.046
SAV.SA, RST.SA	88
PUSH- növeli a stack-et egy szóval a memóriában	604.361
AQPUSH - az A és Q regisztereket betolja a stack memória részére	300.957

RTP szubrutin	hívások száma
APUSH - az A regisztert betolja a stack memória részére	1,204.858
BASE - hivatkozási szint operandus alapján a többszörös indirekt hivatkozást hajtja végre	306.821
DVI.SAQ	7.099
MPI.SAY	21
MPI.SAQ	11.730
GRT.SA	73
GRT.SAQ	216.644
EØL.SAQ	180.043
LES.SAQ	351.137
NØT.S	149.644
MØV.S	42.293
LAD.S	3.187
LAD.SA	0
LØD.S	12.823
LØD.SA	1.254
STØS.SAQ	147
EQLM.S	142.818
LESM.S	87.861
NEQM, GRTM, GEQM, LEQM, EQLS, LEQS, GEQS (minden VSS mellett)	0
NEQS.SAQ	2
NEQS.S, NEQS.SA	0

F.4 Tapasztalati adatok, statisztikák

A PASCAL.P forrásprogram lefordításának idő és memória-igénye a PASCAL.P/CDC 3300 cimpiler különböző változataira.

Compiler version	a compiler jellemzője	generált kód	idő igény		memória igény	
			kód generálás	assemblálás	utasitás	adat
0	A lausanni anyagéprõl származó kódcsoportokra optimalizált változat	SC kód	14 perc	21 perc	32K	18K
1	Kódcsoportokra optimalizált, az operációs rendszerhez illesztett, vágott változat	SC kód	8 perc	26 perc	22K	14K
2	Helyi továbbfejlesztés, kódcsop. nem optimalizált, vstack elven müködõ változat	VSC kód	5 perc	22 perc	20K	14K

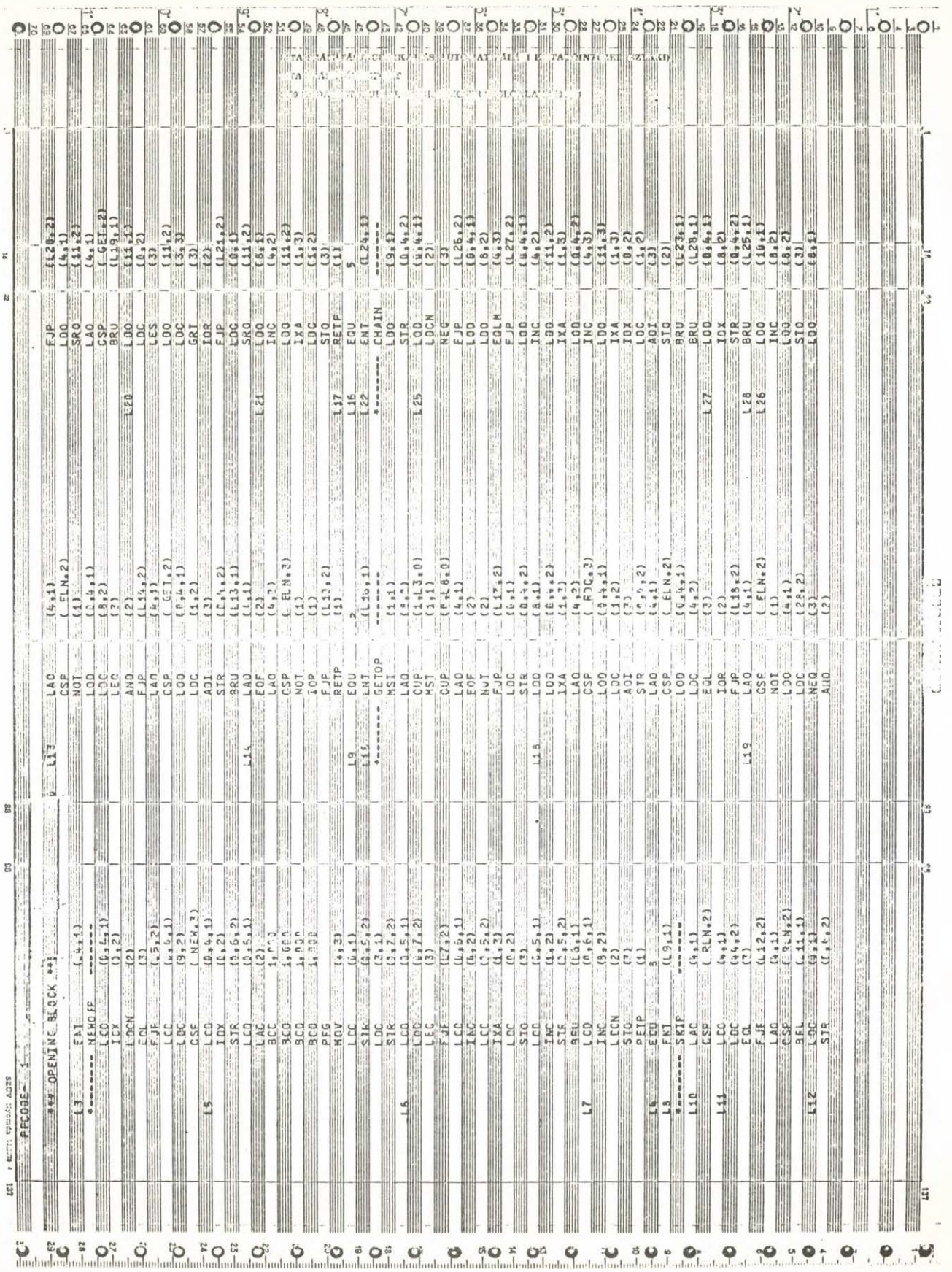
F.5 Példák

1. Az OPSTAT program PASCAL.P nyelven

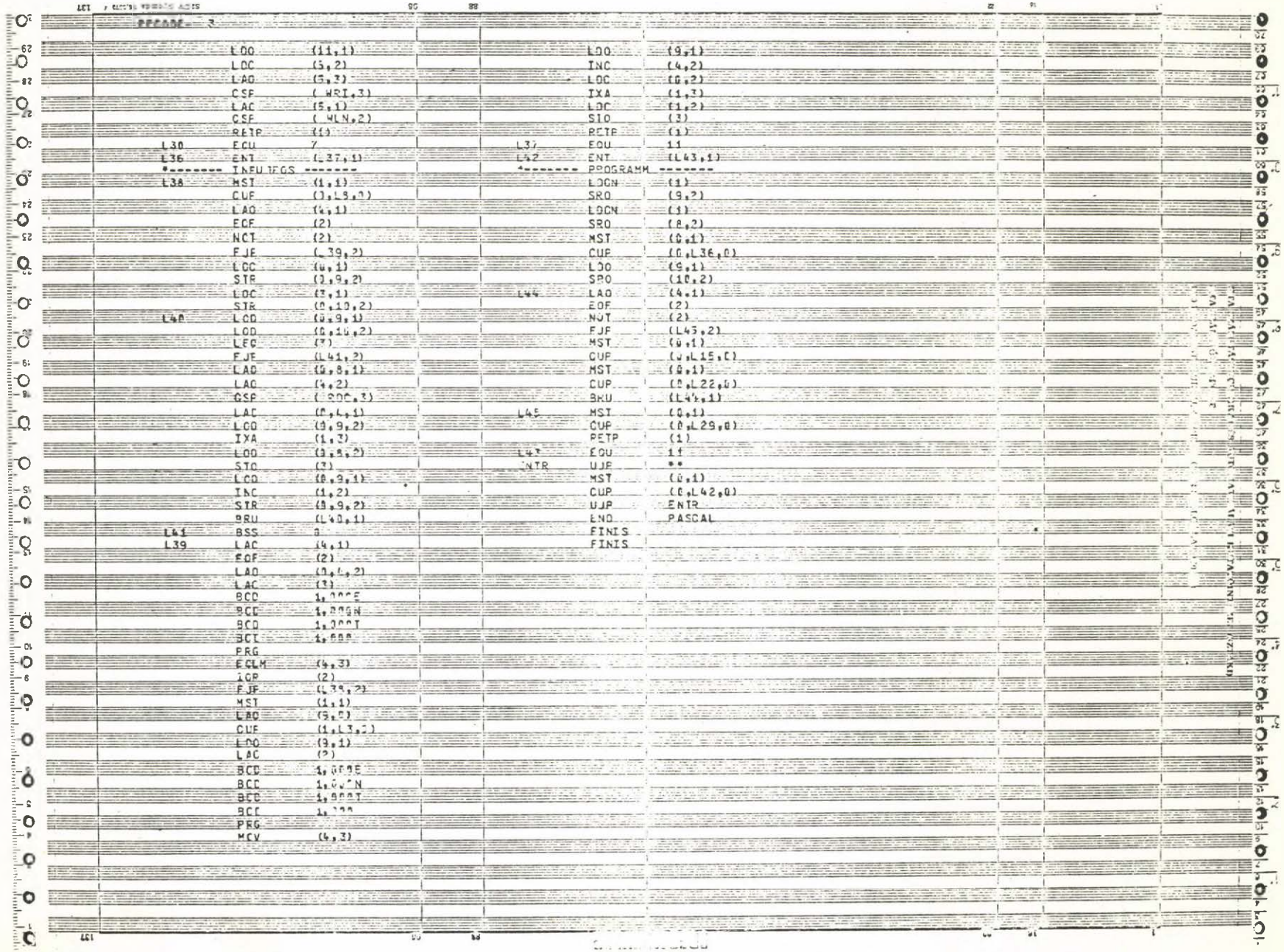
```
PROGRAM OPSTAT(INPUT,OUTPUT);

(*************************************************************************
    PURPOSE -    GIVE A STATISTIC ABOUT VSC-CODE INSTRUCTIONS

    INPUT   -    VSC-CODE (GENERATED BY PASCAL.P/CDC 3300
                 VERSION 2.) IN CARD IMAGES FORM
    CARD IMAGE
       CHAR :    1         10        20
                 +         +         +
       CARD :    *         MNEM      ... ,VSS)
                 +         +         +
       FIELDS :  COMMENT   OPERAND   PARAMETER-LIST

       VSS PARAMETER VALUES :   0 = Z
                                1 = S
                                2 = SA
                                3 = SAO

    OUTPUT  -    THE NUMBER OF OCCURENCES OF EACH INSTRUCTION WITH
                 SPECIAL  VSS  PARAMETER WILL BE ACCUMULATED AND
                 PRINTED IN A TABLE FORMAT
*************************************************************************)

TYPE
    OPP=^OPRECORD;
    OPRECORD=RECORD
                 MNEM:PACKED ARRAY[0..3]OF CHAR;
                 VSSN:ARRAY[0..3]OF INTEGER;
                 NEXT:OPP
             END;
VAR LOPP,COPP,NOWP :OPP;
    V:INTEGER;

    PROCEDURE NEWOPP(VAR XOPP:OPP);
    VAR J:INTEGER;
    BEGIN
        IF XOPP=NIL THEN
            NEW(XOPP);
            WITH XOPP^ DO
            BEGIN
                MNEM:='    ';
                FOR J:=0 TO 3 DO VSSN[J]:=0;
                NEXT:=NIL
            END
    END;

        PROCEDURE SKIP;
        VAR K:INTEGER;
        BEGIN
        REPEAT
            READLN;
            WHILE INPUT^='*' DO
            READLN;                          (*SKIP COMMENT CARDS*)
            K:=0;
            WHILE (NOT EOLN) & (K<=8) DO
            BEGIN GET(INPUT); K:=K+1 END     (*SKIP LABEL FIELD*)
        UNTIL EOF OR NOT EOLN
        END;

    PROCEDURE GETOP;
    VAR K:INTEGER;
```

```
 63      4
 64      6       BEGIN
 65     76           NEWOPP(NOWP);
 66     80           SKIP;
 67     82           IF NOT EOF THEN
 68     82           BEGIN
 69     86                K:=1;
 70     86                REPEAT
 71     88                READ(NOWP↑.MNEM[K]); K:=K+1
 72     94                UNTIL EOLN ∨ (K=4);
 73    104            WHILE (NOT EOLN) ∧ (INPUT↑ <> ↓)↓ DO
 74    112            BEGIN
 75    112              V:= ORD(INPUT↑); GET(INPUT);
 76    116            END;
 77    117              IF (V<0) ∨ (V>3) THEN V:=0;        (*DEFAULT VSS VALUE = 0 *)
 78    127            NOWP↑.VSSN[V]:=1
 79    131            END;
 80    133       END;
 81    134
 82    134
 83    134       PROCEDURE CHAIN;
 84    134       LABEL 76;
 85      4       VAR YOPP,OPP:
 86      5
 87      6       BEGIN
 88    134           YOPP:=OPP;
 89    137           WHILE YOPP<>NIL DO
 90    141           BEGIN
 91    141                IF  YOPP↑.MNEM=NOWP↑.MNEM  THEN
 92    145                  BEGIN YOPP↑.VSSN[V]:= YOPP↑.VSSN[V]+1;
 93    157                  GOTO 76
 94    158                  END
 95    158                ELSE YOPP:= YOPP↑.NEXT
 96    166           END;
 97    163           LOPP↑.NEXT:=NOWP;
 98    167           LOPP:=NOWP; NOWP:=NIL;
 99    169       76:
100    171       END;
101    172
102    172
103    172       PROCEDURE CHAINWRITE;
104    172       VAR S,I:INTEGER;
105      6
106      6       BEGIN
107    172            WRITE(↓MNEMONIC↓:11);
108    178            WRITE(↓I↓:6,↓S↓:6,↓SA↓:6,↓SAO↓:6);
109    196            WRITELN(↓SUMMA↓:12);
110    203            WRITELN;
111    205            V:=1;
112    205           WHILE  OPP<>NIL  DO
113    211           BEGIN
114    211                 S:=0;
115    211               IF (OPP↑.MNEM <> ↓    ↓)
116    214                ∧ (OPP↑.MNEM <> ↓FIN↓)
117    217                ∧ (OPP↑.MNEM <> ↓END ↓)
118    221                ∧ (OPP↑.MNEM <> ↓UJP ↓)
119    225                                          (*EXCLUDE NO VSS INSTRUCTIONS*)
120    227                                        THEN
121    229               BEGIN
122    229                  WRITE(OPP↑.MNEM:11);
123    234                  FOR I:=0 TO 3 DO
124    242                     BEGIN
125    242                     WRITE(OPP↑.VSSN[I]:6);
126    250                     S:=S+OPP↑.VSSN[I]
127    255                     END;
128    258                  WRITELN(↓ ↓:6,S:6);
129    273               END;
130    273                  OPP:=OPP↑.NEXT;
```

2. Az OPSTAT program VSC kódja



```
PROGR= 2

                SRO     (10,2)                  CSP     (_WLN,2)                LOD     (0,4,2)
                LDCN    (1)                     LDC     (4,1)                   IXA     (1,3)
                SRO     (3,2)                   SRO     (11,2)                  IDX     (0,2)
        L23     EQU     (1)             L31     LDO     (9,1)                   LDC     (6,2)
        L24     EQU     5                       LDCN    (2)                     LAO     (5,3)
        L29     ENT     (L30,1)                 NEQ     (7)                     CSP     (_WRI,3)
        ******** CHAINMPI ********              FJP     (L32,2)                 LOD     (0,5,1)
                LAC     (1)                     LDC     (0,1)                   LDO     (9,2)
                BCD     1,0004                  STR     (0,5,2)                 INC     (4,3)
                BCD     1,0000                  LDO     (9,1)                   LOD     (0,4,3)
                BCD     1,0005                  LAC     (2)                     IXA     (1,3)
                BCD     1,0004                  BCD     1,000                   IDX     (0,2)
                BCD     1,0000                  BCD     1,000                   AOI     (2)
                BCD     1,0001                  BCD     1,000                   STR     (0,5,1)
                BCD     1,0001                  BCD     1,000                   LOD     (0,4,1)
                BCD     1,0000                  PRG                             INC     (1,2)
                PFG                             NEQM    (4,3)                   STR     (0,4,2)
                LDC     (14,2)                  LDO     (9,2)                   BRU     (L34,1)
                LDC     (3,3)                   LAC     (3)             L35     LAC     (1)
                LAO     (5,3)                   BCD     1,000F                  BCD     1,000=
                CSP     (_WRS,3)                BCD     1,0001                  BCD     1,000*
                LDC     (57,1)                  BCD     1,000A                  PRG
                LDC     (5,2)                   BCD     1,0001                  LDC     (6,2)
                LAO     (5,3)                   PRG                             LDC     (2,3)
                CSP     (_WRS,3)                NEQM    (4,3)                   LAO     (5,3)
                LDC     (54,1)                  AND     (2)                     CSP     (_WRS,3)
                LDC     (3,2)                   LDO     (9,2)                   LOD     (0,5,1)
                LAO     (5,3)                   LAC     (3)                     LDC     (6,2)
                CSP     (_WRS,3)                BCD     1,000E                  LAO     (5,3)
                LAC     (1)                     BCD     1,000A                  CSP     (_WRI,3)
                BCD     1,0005                  BCD     1,000C                  LAO     (5,1)
                BCD     1,000A                  BCD     1,000                   CSP     (_WLN,2)
                PPG                             PRG             L33     LDO     (9,1)
                LDC     (5,2)                   NEQM    (4,3)                   IDX     (8,2)
                LDC     (2,3)                   AND     (2)                     SRO     (9,2)
                LAO     (5,3)                   LDO     (9,2)                   LDO     (11,1)
                CSP     (_WRS,3)                LAC     (3)                     LOD     (0,5,2)
                LAC     (1)                     BCD     1,000U                  ADI     (3)
                BCD     1,0005                  BCD     1,000J                  SRO     (11,2)
                BCD     1,000A                  BCD     1,000F                  BRU     (L31,1)
                BCD     1,0000                  BCD     1,000           L32     LAO     (5,1)
                PRG                             PRG                             CSP     (_WLN,2)
                LDC     (6,2)                   NEQM    (4,3)                   LAC     (1)
                LDC     (2,3)                   AND     (2)                     BCD     1,000T
                LAO     (5,3)                   FJP     (L33,2)                 BCD     1,0000
                CSP     (_WRS,3)                LDO     (9,1)                   BCD     1,000T
                LAC     (1)                     LDC     (10,2)                  BCD     1,000A
                BCD     1,0005                  LDC     (4,3)                   BCD     1,000L
                BCD     1,0000                  LAO     (5,3)                   BCD     1,000
                BCD     1,0004                  CSP     (_WRS,3)                BCD     1,000
                BCD     1,0004                  LDC     (0,1)                   BCD     1,000
                BCD     1,000A                  STR     (0,4,2)                 BCD     1,000
                PRG                             LDC     (3,1)                   BCD     1,000=
                LDC     (12,2)                  STR     (0,5,2)                 BCD     1,000=
                LDC     (5,3)           L34     LDO     (0,4,1)                 PRG
                LAO     (5,3)                   LDO     (0,5,2)                 LDC     (40,2)
                CSP     (_WRS,3)                LEQ     (3)                     LDC     (11,3)
                LAC     (3,1)                   FJP     (L35,2)                 LAO     (5,3)
                CSP     (_WLN,2)                LDO     (9,1)                   CSP     (_WRS,3)
                LAO     (5,1)                   INC     (4,2)
```



```
PFCADE = 3

                LOD     (11,1)                          LOD     (9,1)
                LOC     (5,2)                           INC     (4,2)
                LAO     (5,7)                           LOC     (0,2)
                CSP     ( WRI,3)                        IXA     (1,3)
                LAC     (5,1)                           LOC     (1,2)
                CSP     ( WLN,2)                        STO     (3)
                RETP    (1)                             RETP    (1)
        L30     EQU     7               L37             EQU     11
        L36     ENT     (L37,1)         L42             ENT     (L43,1)
        *-------- INDEFINIS -------     *-------- PROGRAMM --------
        L38     MST     (1,1)                           LDCN    (1)
                CUP     (1,L8,2)                        SRO     (9,2)
                LAO     (4,1)                           LDCN    (1)
                EOF     (2)                             SRO     (8,2)
                NOT     (2)                             MST     (0,1)
                FJP     (L39,2)                         CUP     (0,L36,0)
                LOC     (6,1)                           LOD     (9,1)
                STR     (1,9,2)                         SRO     (10,2)
                LOC     (3,1)           L44             LAO     (4,1)
                STR     (0,10,2)                        EOF     (2)
        L40     LOD     (0,9,1)                         NOT     (2)
                LOD     (0,10,2)                        FJP     (L45,2)
                LEQ     (7)                             MST     (0,1)
                FJP     (L41,2)                         CUP     (1,L15,0)
                LAO     (0,8,1)                         MST     (0,1)
                LAO     (4,2)                           CUP     (0,L22,0)
                CSP     (BOOC,3)                        BRU     (L44,1)
                LAC     (0,4,1)         L45             MST     (0,1)
                LOD     (0,9,2)                         CUP     (0,L29,0)
                IXA     (1,2)                           RETP    (1)
                LOD     (0,8,2)         L47             EQU     11
                STO     (3)             ENTR            UJP     **
                LOD     (0,9,1)                         MST     (0,1)
                INC     (1,2)                           CUP     (0,L42,0)
                STR     (0,9,2)                         UJP     ENTR
                BRU     (L40,1)                         END     PASCAL
        L41     BSS     0                               FINIS
        L39     LAC     (4,1)                           FINIS
                EOF     (2)
                LAO     (0,4,2)
                LAC     (3)
                BCD     1,
                BCD     1,
                BCD     1,
                BCT     1,
                PRG
                ECLM    (4,3)
                LOP     (2)
                FJP     (L38,2)
                MST     (1,1)
                LAO     (0,2)
                CUP     (1,L3,1)
                LOD     (0,1)
                LAC     (2)
                BCD     1,
                BCD     1,
                BCD     1,
                BCT     1,
                PRG
                MCV     (4,3)
```

3. Az OPSCAN program PASCAL.P nyelven

```
COSY/MASTER VER 2.4     14/15/76
   OPSCAN   DECK/      I,H,T=80
            ENDCOSY/
REWIND(SHO)
FILE,PIN=SHO
PAS,CAL(I=PIN,G=PRR,L)
    1         6 PROGRAM  OPSCAN(INPUT,OUTPUT);
    2         8
    3         8 CONST    ROOT=4FJ? 4;
    4         8       RINGSIZE=4;
    5         8       SECOND=FALSE;
    6         8 TYPE     HALFA=PACKED ARRAY [0..3] OF CHAR;
    7         8          CRC = ^CIRCULUS;
    8         8          CIRCULUS=RECORD
    9         8                      OP:HALFA;
   10         8                      VSS:INTEGER;
   11         8                      BACK,FORW:CRC;
   12         8                      MARKED: BOOLEAN
   13         8                    END;
   14         8          TRE=^BINTREE;
   15         8          BINTREE=RECORD
   16         8                        NEM : HALFA;
   17         8                      COUNT,VOT: INTEGER;
   18         8                      NEXT,SUBT:TRE
   19         8                    END;
   20         8 VAR CH:CHAR;
   21         8       CRCP:CRC;
   22        10       TREP:TRE;
   23        11       TRTS:ARRAY[0..3] OF  TRE;
   24        15       I:INTEGER;
   25        16
   26        16
   27        16       PROCEDURE ROOTOP;
   28         4
   29         4
   30         4           PROCEDURE SKIP;
   31         4           VAR  K:INTEGER;
   32         4           BEGIN
   33         3           REPEAT
   34         4                 READLN;
   35         6                 WHILE INPUT^=4^4 DO
   36        10                 READLN;                          (*SKIP COMMENT CARDS*)
   37        13                 K:=0;
   38        13                  WHILE (NOT EOLN) ^ (K<=8) DO
   39        21                 BEGIN GET(INPUT); K:=K+1 END     (*SKIP LABEL FIELD*)
   40        29           UNTIL EOF ^ NOT EOLN
   41        32           END;
   42        38
   43        38           PROCEDURE ROTA;
   44        38           BEGIN
   45        38              CRCP:=CRCP^.FORW;
   46        42              WITH CRCP^  DO
   47        44              BEGIN
   48        44                    OP:=4     4;
   49        47                    VSS:=1;
   50        51                    MARKED:=FALSE
   51        53              END
   52        55           END;
   53        56
   54        56            PROCEDURE NEWOP;
   55        56            VAR  K:INTEGER;
   56         5            BEGIN
   57        56                 IF NOT EOF THEN
   58        61                 BEGIN
   59        61                      K:=0;
   60        63                      REPEAT
   61        63                    READ(CRCP^.OP[K]); K:=K+1
   62        69                     UNTIL EOLN ^ (K=4);
```

```
            WHILE (NOT EOLN) ^ (INPUT^ <> ^)^) DO
            BEGIN K:= ORD(INPUT^); GET(INPUT) END;
            IF (K<0) v (K>3) THEN K:=0;
          CROP^.VSS:=K
        END;
    END;

BEGIN(*ROOTOP*)
    REPEAT
    ROTA;
    SKIP;
    NEWOP
    UNTIL (CROP^.OP = ROOT) v EOF
END;

PROCEDURE PUTREE;

    PROCEDURE SEARCH(CROX:CRC);
    LABEL 75; VAR TREY: TRE;

        PROCEDURE INSERT;
        BEGIN
          NEW(TREY);
          WITH TREY^,CROX^ DO
          BEGIN
                MNEM:=OP; VST:=VSS;
                COUNT:=1; SUBT:=NIL;
            NEXT:=TREP^.SUBT
          END;
        TREP^.SUBT:=TREY
        END;

    BEGIN(*SEARCH*)
        TREY:=TREP^.SUBT;
        WHILE TREY <> NIL DO
        BEGIN
        IF CROX^.OP = TREY^.MNEM THEN
            BEGIN TREY^.COUNT := TREY^.COUNT+1; GOTO 75 END
            ELSE TREY := TREY^.NEXT
        END;
        INSERT;
     75: TREP:=TREY
    END;

BEGIN(*PUTREE*)
    IF NOT EOF THEN
    BEGIN
     CROP^.MARKED:=TRUE;
     I:=CROP^.VSS;
     IF SCODE THEN I:=0;
     TREP:=TRTS[I];
     TREP^.COUNT:=TREP^.COUNT+1;
     WHILE NOT CROP^.BACK^.MARKED DO
     BEGIN
     CROP:=CROP^.BACK ;
     SEARCH(CROP);
     CROP^.MARKED:=TRUE
     END
    END
END;

PROCEDURE TREEWRITE;
VAR MARRAY[1..RINGSIZE]OF HALFA;
    CARRAY[1..RINGSIZE]OF INTEGER;
     TOTAL,V :INTEGER;
     MN:HALFA;
```

```
        PROCEDURE WRITEPROC(I:INTEGER);
        VAR  D,K:INTEGER;
         BEGIN
              IF C[I] < 0 THEN WRITE_N('*** ERROR ***',20);
              IF C[I] > 0 THEN
              BEGIN
                   WRITE(C[I]:10);
                   D:=C[I];
                   TOTAL:=TOTAL+D;
                   FOR K:=I TO RINGSIZE DO
                   BEGIN
                   WRITE(M[K]:10);
                   C[K]:=C[K]-D;
                   END;
              WRITELN('.',V:1,V:1);
              END
         END;

        PROCEDURE TREEWI(XP:TREE;I:INTEGER);
         BEGIN   IF  XP<>NIL  THEN
              BEGIN
              M[I]:=XP^.MNEM;
              C[I]:=XP^.COUNT;
              TREEWI(XP^.SUBT,I-1);
              WRITEP(OC(I));
              TREEWI(XP^.NEXT,I)
              END
         END;

    BEGIN(*TREEWRITE*)
         TOTAL:=0;
         WRITE('COUNT',13);
         WRITE('INSTRUCTION GROUPS OF LENGTH <= ',35);
         V:=RINGSIZE;
         WRITE(V:1);
         WRITE(' , ENDED BY ',13);
         MN:=ROOT;
         WRITELN(MN:5);
                   FOR V:=0 TO 3 DO
         BEGIN
              WRITELN;
              TREEWI(TRTS[V],RINGSIZE);
              WRITELN
         END;
         WRITELN(' TOTAL=',1,TOTAL:4)
    END;


    PROCEDURE INITS;

        PROCEDURE INITREC;
        VAR  I:INTEGER;
        BEGIN
          FOR I:=0 TO 3 DO
          BEGIN
               NEW(TRTS[I]);
               WITH TRTS[I]^ DO
               BEGIN
               SUBT:=NIL;
               NEXT:=NIL;
               MNEM:=ROOT;
               COUNT:=0;
               VST:=1
               END
          END
        END;
```

```
        PROCEDURE INITCIRCUS(VAR CRCX:CRC);
        VAR  CRCY:CRC;
        BEGIN
           NEW(CRCY);
           IF CRCX=NIL THEN CRCX:=CRCY;
           CRCY^.FORW:=CRCX;
           CRCY^.BACK:=CRCX;
           CRCX^.FORW^.BACK:=CRCY;
           CRCY^.FORW:=CRCX^.FORW;
           CRCX^.FORW:=CRCY;
           CRCX:=CRCY
        END;
    BEGIN(*INITS*)
        INITREE;
        CRCP:=NIL;
        FOR I:=1 TO RINGSIZE DO INITCIRCUS(CRCP)
    END;

BEGIN(*UPSCAN*)
        INITS;
        WHILE NOT EOF DO
        BEGIN
           ROUTOP;
           PUTREE
        END;
        TREEWRITE
END.
```

* * * * * WARNING,THIS IS LAST PAGE OF OUTPUT

IRODALOMJEGYZÉK

[1] N. Wirth, "The Programming Language PASCAL", ACTA INFORMATICA, 1, 35-63, (1971).

[2] K. Jensen, N. Wirth, PASCAL User Manual and Report, Lecture Notes in Computer Sciences, Springer Verlag, 1974.

[3] N. Wirth, "Program Development by Stepwise Refinement", Communications ACM,14, 221-227, April, 1971.

[4] D.J. Dahl, E.W. Dijkstra, C.A.R. Hoare, Structured Programming, Academic Press Inc. 1972.

[5] K.V. Nori, V. Amman, K.Jensen, H.H. Nägeli The Pascal <P> Compiler Implementation Notes Berichte des instituts für Informatike

[6] L. Ammeraal: Extending a runtime stack with some registers Afelding Informatica IW 48/75 september

[7] PASCAL.P/CDC 3300 Felhasználói kézikönyv (printer listán készül)