Ambiance de execuție

În etapa premergătoare generării de cod trebuie realizate o serie de activități care implementează legăturile între textul static al programului sursă cu acțiunile care urmează să aibă loc la execuție. De exemplu, același nume din textul sursă poate reprezenta în execuție obiecte de date diferite iar codul obiect generat trebuie să le diferențieze fără echivoc.

Alocarea și relocarea obiectelor de date este realizată de **suportul de execuție**. Acesta constă într-o colecție de rutine încărcate împreună cu codul obiect generat. Tehnicile prezentate în continuare sunt utile la proiectarea unui astfel de suport pentru limbaje de programare ca Fortran, Pascal, C sau Lisp.

Fiecare execuție a unei proceduri se numește **activare** a procedurii. Dacă o procedură este recursivă, pot fi în execuție, în același timp, mai multe activări ale sale; fiecare activare în parte manipulează obiecte de date alocate special pentru ea.

Reprezentarea unui obiect de date în timpul execuției este conformă tipului său. Astfel, datele elementare (caractere, valori întregi sau reale) au reprezentări echivalente în mașina de calcul și se reprezintă direct prin acestea. Ansambluri de date cum sunt tablourile, șirurile sau structurile sunt reprezentate prin colecții de obiecte primitive cu o anumită dispunere în memorie de care trebuie să se țină cont la selectarea componentelor.

Probleme specifice derivate din structura și conținutul limbajului sursă

Se presupune că limbajul sursă include proceduri în forma din Pascal. Trebuie făcută distincție între textul sursă al unei proceduri și activările sale la execuție.

Proceduri

O **definiție de procedură** este o declarație care, în forma sa cea mai simplă, asociază un identificator cu o instrucțiune. Identificatorul este **numele procedurii** iar instrucțiunea este **corpul procedurii**.

Să considerăm următorul exemplu de program Pascal care citește și sortează un șir de numere întregi:


```
(1)  program Sort;
(2)      var a: array [0 .. 10] of integer;
(3)      procedure CitesteSir;
(4)          var i: integer;
(5)          begin
(6)              for i:=1 to 9 do read (a[i])
(7)          end;
(8)      function Partitionare (y, z: integer) : integer;
(9)          var i, j, x, v: integer;
(10)         begin …
(11)         end;
(12)     procedure QuickSort (m, n: integer);
(13)         var i: integer;
(14)         begin
(15)             if (n>m) then begin
(16)                 i:=Partitionare (n, m);
(17)                 QuickSort (m, i-1);
(18)                 QuickSort (i+1, m);
(19)             end;
(20)         end;
(21)  begin
(22)         a[0]:=-9999; a[10]:=9999;
(23)         CitesteSir;
(24)         QuickSort (1, 9)
(25)  end.
```

Subprogramele care returnează valori prin numele lor sunt numite în mod uzual **funcții**. În cele ce urmează ele vor fi referite într-un mod similar procedurilor cu deosebirea că apelurile de funcții pot apărea în expresii, ca în linia 16. De asemenea și programul complet va fi tratat ca procedură. În programul exemplu sunt ilustrate principalele noțiuni legate de conceptul de procedură:

- definiția procedurii;
- apelul său;
- parametrii formali (în C – argumente formale iar, în Fortran – argumente fictive);
- argumentele (parametrii actuali).

Metodele de stabilire a corespondenței între parametrii formali și argumente sunt specifice limbajului de programare și vor fi discutate ulterior, în mod distinct.

Arbori de activare

În legătură cu fluxul comenzilor între proceduri pe parcursul execuției programului, se fac următoarele presupuneri:

1) Comenzile se execută secvențial: execuția unui program constă dintr-o secvență de pași, la fiecare pas controlul fiind într-un punct anume din program.

2) Fiecare execuție a unei proceduri debutează cu începutul corpului procedurii și, la sfârșit, returnează controlul la punctul imediat următor locului în care procedura a fost apelată ⇒ Fluxul controlului între proceduri poate fi ilustrat prin arbori.

Aşa cum s-a arătat, fiecare execuție a unui corp de procedură se numeşte activare a procedurii. **Durata de viață** a unei activări a procedurii **p** este intervalul de timp consumat pentru execuția corpului procedurii, inclusiv timpul petrecut pentru executarea procedurilor apelate de **p**, a procedurilor apelate de acestea ş.a.m.d. În general, termenul de "durată de viață" se referă la o secvență consecutivă de paşi în timpul execuției unui program.

În limbajele algoritmice uzuale, ca Pascal de exemplu, de fiecare dată când controlul intră într-o procedură "q", apelată din procedura "p", el se va întoarce în final la "p" (în absența unei erori fatale). Mai exact, de fiecare dată când comanda trece de la o activare a procedurii "p" la o activare a procedurii "q", ea se va întoarce la aceeaşi activare a procedurii "p".

Dacă **a** şi **b** sunt activări de proceduri atunci duratele lor de viață sunt fie disjuncte fie suprapuse (încuibate): dacă se intră în **b** înainte de a părăsi **a**, atunci controlul trebuie să părăsească **b** înainte de a părăsi **a**. Această proprietate de încuibare a duratei de viață a activărilor poate fi ilustrată prin inserarea a două instrucțiuni de tipărire în fiecare procedură, una înainte de prima instrucțiune a corpului procedurii iar cealaltă după ultima instrucțiune. Prima tipăreşte **intrare** "nume procedură (argumente)" iar ultima tipăreşte **ieşire** "nume procedură (argumente)". Execuția programului **Sort** din exemplul anterior, cu aceste tipăriri incluse, determină afişarea următoarelor informații:

Inceputul executiei
  intrare CitesteSir
  iesire CitesteSir
  intrare QuickSort (1, 9)
  intrare Partitionare (1, 9)
  iesire Partitionare (1, 9)
  intrare QuickSort (1, 3)
……
  iesire QuickSort (1, 3)
  intrare QuickSort (5, 9)
……
  iesire QuickSort (5, 9)
  iesire QuickSort (1, 9)
Terminarea executiei

De exemplu, durata de viață a activării "QuickSort (1, 9)" este secvența de paşi executați între tipărirea mesajului "**intrare** QuickSort (1, 9)" şi cea a mesajului "**iesire** QuickSort (1, 9)". În ceea ce priveşte procedura (funcția) **Partitionare**, s-a presupus că valoarea returnată de apelul "Partitionare (1,9)" este **4**.

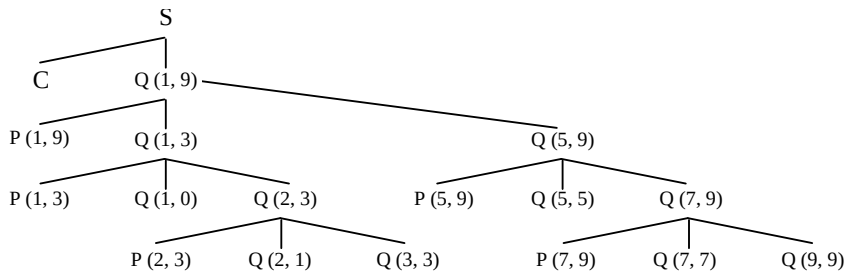
În termenii de mai sus, o procedură este **recursivă** dacă o nouă activare a ei poate începe înainte de încheierea unei activări anterioare. Exemplu: **QuickSort**, de mai sus. Apelul unei proceduri recursive poate fi direct (**p** apelează **p**) sau poate fi indirect (**p** apelează **q** care apelează, apoi, **p**).

Modul în care fluxul de control intră şi iese din activările procedurilor poate fi ilustrat printr-un arbore numit **arbore de activare**, având următoarele proprietăți:

1) Fiecare nod reprezintă o activare a unei proceduri;
2) Rădăcina reprezintă activarea programului principal;
3) Nodul pentru **a** este părintele nodului pentru **b** dacă şi numai dacă fluxul de control trece de la activarea lui **a** la **b**;
4) Nodul pentru **a** este la stânga nodului pentru **b** dacă şi numai dacă durata de viață a lui **a** începe înaintea duratei de viață a lui **b**.

Deoarece fiecare nod reprezintă o unică activare şi invers ⇒ controlul se află la un nod atunci când el este în activarea reprezentată de acel nod.

Exemplu: Arborele de activare corespunzător execuției programului **Sort** este următorul:



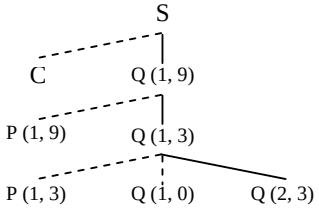
N-a fost ilustrată decât partea superioară a arborelui. Rădăcina corespunde întregului program **Sort**. În timpul execuției lui **Sort** există o activare a lui CitesteSir reprezentată de fiul rădăcinii cu eticheta C. Următoarea activare, reprezentată de al doilea fiu al rădăcinii, este pentru QuickSort cu argumentele 1 şi 9. În timpul acestei activări, apelurile la Partitionare şi QuickSort conduc la activările P(1, 9), Q(1, 3) şi Q(5, 9). Se observă că activările Q(1, 3) şi Q(5, 9) sunt recursive: ele încep înainte ca activarea Q(1, 9) să se încheie.

Stive de control

Fluxul controlului într-un program corespunde unei traversări spre adâncime a arborelui de activare corespunzător: începe de la rădăcină, vizitează un nod părinte înaintea fiilor săi iar fiii fiecărui nod sunt vizitați recursiv, de la stânga la dreapta. Prin traversarea în acest fel a arborelui de activare, tipărind **intrare …** când nodul pentru o activare a fost atins prima dată şi tipărind **iesire …** la revenire, după ce a fost vizitat întregul subarbore al nodului, se poate reconstitui fişierul de afişare pentru execuția programului **Sort**.

Pentru a ține evidența activărilor **viabile** de proceduri, se poate utiliza o stivă, numită **stivă de control**. Principiul de lucru cu stiva este următorul: nodul pentru o activare se introduce în stivă când începe activarea şi se extrage la terminarea ei. ⇒ Conținutul stivei de control este legat de căile la rădăcina arborelui de activare: când un nod **n** este în vârful stivei de control, stiva conține nodurile pe calea de la **n** la rădăcină.

Exemplu: Referindu-ne tot la programul **Sort** şi la arborele de activare corespunzător, în momentul în care controlul intră în activarea reprezentată de nodul Q(2, 3), nodurile din arborele de activare care au fost deja atinse sunt:



Activările cu etichetele C, P(1, 9), P(1, 3), Q(1, 0) au fost executate şi terminate, legăturile lor în arbore fiind marcate cu o linie întreruptă. Liniile continue marchează drumul de la Q(2, 3) la rădăcină. În acest punct, stiva de control conține următoarele noduri (vârful stivei este la dreapta):

S	Q(1, 9)	Q(1, 3)	Q(2, 3)	

↑
v_s

Stivele de control sunt utile la implementarea unor limbaje de programare ca Pascal sau C, pentru tehnicile de alocare a memoriei prin stivă.

Domeniul unei declarații

O declarație este o construcție sintactică a unui limbaj prin care unui nume i se asociază alte informații. Declarațiile pot fi explicite, ca în exemplul de mai jos (Pascal):

var i: integer;

sau pot fi implicite, ca în Fortran, unde *I* va desemna o variabilă întreagă, dacă nu se declară altfel în mod explicit.

În diferite părți ale unui program pot exista mai multe declarații, independente, ale aceluiaşi nume. Stabilirea declarației valabile într-un anumit punct din textul programului depinde de **regulile de domeniu** ale limbajului respectiv. De exemplu, în programul Pascal **Sort**, **i** este declarat de trei ori, în liniile 4, 9 şi 13 iar utilizările numelui **i** în procedurile **CitesteSir**, **Partitionare** şi **QuickSort** sunt independente una de alta ⇒ cele două apariții ale lui **i** în linia 6 sunt în domeniul declarației din linia 4 ş. a. m. d.

Porțiunea programului la care se aplică o declarație se numeşte **domeniul** acelei declarații. O apariție a unui nume într-o procedură se spune că este **locală** procedurii dacă este în domeniul unei declarații din cadrul procedurii; altfel, se spune că apariția este **nelocală**. Distincția între numele locale şi nelocale se aplică la orice construcție sintactică ce poate conține în ea declarații.

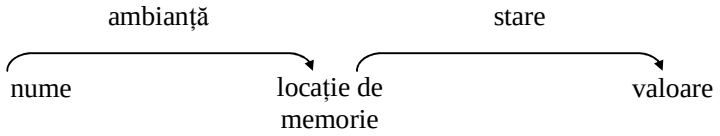
Deşi domeniul este o proprietate a declarației unui nume, **x**, frecvent se utilizează abrevierea "domeniul numelui **x**". De exemplu, domeniul lui **i** din linia 17 a programului **Sort** este corpul procedurii **QuickSort**.

La compilare, pentru a găsi declarația care se aplică unei apariții a unui nume, se va utiliza Tabela de simboluri (TS). Când se întâlneşte o declarație, se creează pentru ea o intrare în TS. Cât timp, pe parcursul compilării, suntem în domeniul declarației, la căutarea, în tabelă, a numelui asociat declarației, se va returna această intrare.

Legarea numelor

Un nume poate denumi obiecte de date diferite în timpul execuției, chiar dacă fiecare nume este declarat numai o dată într-un program. Termenul informal de “obiect de date” corespunde unei locații de memorie care poate conține valori.

În semantica limbajelor de programare, termenul de **ambianță** se referă la o funcție care pune în corespondență un nume cu o locație de memorie iar termenul **stare** se referă la o funcție care pune în corespondență o locație de memorie cu valoarea înmagazinată în ea. Aceste corespondențe în două etape ale numelui cu valoarea corespunzătoare sunt ilustrate în figura de mai jos:



Utilizând termenii cunoscuți de **valoare-stânga** și **valoare-dreapta**, o ambianță pune în corespondență un nume cu o valoare-stânga iar o stare pune în corespondență valoarea-stânga cu o valoare-dreapta.

Ambianțele și stările sunt diferite; o asignare schimbă starea dar nu și ambianța.

Când o ambianță asociază o locație de memorie **s** cu un nume **x**, se spune că **x** este **legat** la **s**; asocierea însăși se numește **legarea** lui **x**. Termenul de “locație” de memorie trebuie luat în sens figurat: dacă tipul lui **x** nu este un tip de bază, memoria **s** pentru **x** poate fi o colecție de cuvinte de memorie.

O legare este corespondentul dinamic al unei declarații, ca în exemplele din tabelul de mai jos:

Noțiunea Statică	Corespondența Dinamică
definiția unei proceduri	activările procedurii
declararea unui nume	legările numelui
domeniul unei declarații	durata de viață a legăturii

Exemple: Mai multe activări ale unei proceduri recursive pot fi viabile în același timp. De asemenea, în Pascal, un nume de variabilă locală este legat la diferite locații de memorie, în fiecare activare a procedurii.

Rezumat al problemelor ce țin de limbajul de programare și influențează implementarea ambianței de execuție

Modul în care un compilator pentru un limbaj trebuie să-și organizeze memoria și să lege numele este determinat, în mare măsură, de răspunsurile la următoarele întrebări:

1) Procedurile se pot apela recursiv?

2) Ce se întâmplă cu valorile numelor locale când controlul revine dintr-o activare a unei proceduri?

3) Un nume nelocal poate fi referit într-o procedură?

4) Cum se transferă parametrii la apelul unei proceduri?

5) Procedurile pot fi transmise ca parametri?

6) Numele de proceduri pot fi returnate ca rezultate?

7) Memoria poate fi alocată dinamic, sub controlul programului?

8) Se poate face relocarea explicită a memoriei?

În continuarea acestui capitol se va prezenta efectul acestor probleme asupra suportului de execuție necesar pentru implementarea unui limbaj de programare dat.

Organizarea memoriei

Problemele privind organizarea memoriei la execuție au o legătură strânsă cu unele facilități ale limbajului de programare. Soluțiile tratate în acest subcapitol sunt potrivite pentru limbaje algoritmice de genul Fortran, Pascal sau C.

Subîmpărțirea memoriei de execuție

În principiu, zona de memorie necesară execuției unui program compilat poate fi împărțită, în funcție de destinația părților sale, astfel:

1) codul obiect generat de compilator;
2) obiectele de date ale programului;
3) parte corespunzătoare a *stivei de control*, necesară pentru evidența activărilor procedurilor.

Dimensiunea codului obiect generat este fixată la compilare ⇒ zona respectivă poate fi determinată static şi plasată, de exemplu, la începutul memoriei disponibile pentru execuție. Similar, şi dimensiunile unora dintre obiectele de date pot fi calculate la compilare ⇒ şi acestea pot fi plasate într-o zonă determinată static, ca în figura de mai jos:

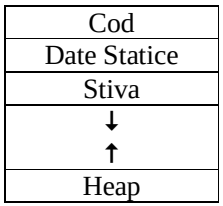


Fig. Împărțire tipică a memoriei de execuție în zone pentru cod obiect şi pentru date (statistice şi dinamice)

Deoarece adresele obiectelor statice pot fi completate direct în codul destinație, simplificând accesul la aceste obiecte în timpul execuției, există chiar un "interes" în favoarea alocării statice a cât mai multor obiecte ale programului. Unele limbaje (ex. Fortran) permit ca toate obiectele de date să fie alocate static.

Implementarea apelurilor recursive determină ca în alte limbaje (ex. Pascal, C) să fie necesară gestionarea explicită a apelurilor procedurilor sub forma unei extensii a stivei de control (stiva din figura anterioară). La apariția unui apel, se suspendă execuția activării curente şi se salvează în stivă informații despre starea maşinii: valorile numărătorului de instrucțiuni şi ale registrelor. La revenirea din apel, se restaurează valorile registrelor, se poziționează numărătorul de instrucțiuni la valoarea imediat următoare apelului şi se continuă activarea suspendată. Obiectele de date ale căror durate de viață sunt conținute în cea a unei activări, pot fi alocate în stivă, împreună cu celelalte informații asociate cu activarea.

Memoria pentru execuție poate să includă şi o zonă distinctă numită **heap** în care, printre altele, se înmagazinează datele dinamice: datele a căror alocare şi eventual relocare se efectuează în mod explicit, prin programul utilizator. Tot în heap se pot păstra informații despre alocările procedurilor în implementările acelor limbaje în care duratele de viață ale activărilor nu pot fi reprezentate sub formă arborescentă. Modul controlat şi simplu în care datele sunt alocate şi relocate în stivă, face ca gestiunea datelor pe principiul stivei să fie mai eficientă decât gestionarea heap-ului.

Reprezentarea creşterii stivei şi a heap-ului ca în figura anterioară (una spre cealaltă) permite ca dimensiunile lor să se poată modifica pe parcursul execuției programului, după necesități. Unele limbaje, cum sunt Pascal sau C, au nevoie la execuție atât de stivă cât şi de heap. Prin convenție, stivele cresc în jos, adică spre adrese mari. Din motive de eficiență a implementării, vârful stivei, notat de obicei cu **top**, se păstrează într-un registru iar adresele în stivă se reprezintă ca deplasamente față de top.

Înregistrări de activare

Informațiile necesare pentru o singură execuție a unei proceduri sunt organizate într-un bloc contiguu de memorie numit **înregistrare de activare** sau **cadru**, având conținutul prezentat în figura de mai jos:

valoarea returnată
argumente
legătura opțională de control
legătura opțională de acces
starea salvată a mașinii
date locale
date temporare

Fig. Structura generală a unei înregistrări de activare

Nu orice implementare de limbaj utilizează toate aceste câmpuri. Unele dintre ele se pot înmagazina în registre. Pentru limbaje ca Pascal sau C, înregistrarea de activare se creează şi se introduce în stiva de execuție la apelul procedurii iar extragerea din stivă şi desființarea ei se realizează la încheierea apelului, atunci când controlul revine la apelant.

Rolul câmpurilor unei înregistrări de activare este următorul (de jos în sus):

1) **Date temporare**: locații pentru valori temporare care apar în procesul de calcul din procedură (de exemplu la evaluarea expresiilor).

2) **Date locale**: datele care sunt locale unei execuții a procedurii.

3) **Starea salvată a maşinii**: informații despre starea maşinii imediat înainte de apelul procedurii; include valorile contorului de program şi ale registrelor, valori care trebuie restaurate atunci când controlul revine din alt apel de procedură.

4) **Legătura opțională de acces**: se utilizează pentru referirea datelor nelocale, conținute în alte înregistrări de activare. Această legătură nu este necesară în limbaje ca Fortran deoarece datele nelocale se păstrează într-un loc fix.

5) **Legătura opțională de control**: indică spre înregistrarea de activare a apelantului.

6) **Argumentele**: sunt utilizate de apelant pentru a furniza parametrii procedurii apelate. Pentru eficiența accesului, în practică, parametrii pot fi înmagazinați şi în registre.

7) **Valoarea returnată**: este utilizată de procedura apelată pentru a returna procedurii apelante o valoare (apel de funcție). Pentru creşterea eficienței şi această valoare poate fi returnată într-un registru.

În mod normal, dimensiunile acestor câmpuri se pot determina în timpul compilării. Ele sunt necesare în momentul execuției, atunci când este apelată procedura. Un caz mai deosebit apare dacă procedura are un tablou local a cărui dimensiune este determinată de valoarea unui argument disponibil, la execuție, abia în momentul apelului.

Organizarea datelor locale la compilare

Presupunem că memoria de execuție este disponibilă în blocuri contigue de octeți. Un anumit număr de octeți formează un cuvânt-calculator. Obiectele multioctet sunt memorate în octeți consecutivi şi primesc adresa primului octet.

Cantitatea de memorie necesară pentru un **nume** este determinată de tipul său. Un tip de date elementar (caracter, întreg, real) poate fi memorat într-un număr întreg de octeți. Memoria pentru o structură de date (tablou, înregistrare) trebuie să înmagazineze toate componentele sale. Pentru acces uşor la componente, este de dorit ca şi zona de memorie pentru o structură să fie alocată într-un bloc contiguu de octeți.

Zona pentru datele locale este organizată la compilare, pe măsură ce se examinează declarațiile dintr-o procedură. Datele de lungime variabilă sunt păstrate în afara acestei zone. Se păstrează un contor al locațiilor de memorie care au fost alocate pentru declarațiile precedente. Din contor se determină adresa relativă a memoriei pentru un obiect local în raport cu începutul zonei sau cu începutul înregistrării de activare.

Organizarea memoriei pentru obiectele date este puternic influențată de limitările de adresare ale calculatorului destinație. De exemplu, instrucțiunile pentru adunarea întregilor ar putea impune ca întregii să fie **aliniați**, adică să fie plasați la anumite poziții în memorie astfel încât adresele lor să fie divizibile cu 4. Din considerente de aliniere, pentru un tablou de 10 caractere s-ar putea aloca 12 octeți (în loc de 10 octeți necesari), lăsând doi octeți neutilizați. Spațiul rămas neutilizat din considerente de aliniere se numeşte "umplutură". Când memoria disponibilă pentru execuție este resursă critică, compilatorul poate **împacheta** datele astfel încât să nu rămână "umplutură". Împachetarea ar putea necesita executarea unor instrucțiuni suplimentare care să poziționeze datele împachetate astfel încât să se poată opera asupra lor ca şi când ar fi aliniate corespunzător.

Exemplu: În tabelul de mai jos se prezintă în mod simplificat organizarea datelor utilizată de compilatoare C pentru două calculatoare numite **Calc.1** şi **Calc.2**. C permite întregi de trei dimensiuni, declarați cu cuvintele cheie **short**, **int** şi **long**. Setul de instrucțiuni pentru cele două calculatoare impune ca în Calc.1 să se aloce 16, 32 şi 32 de biți pentru cele trei tipuri de întregi iar în Calc.2 se vor aloca 24, 48 şi respectiv 64 de biți. Cu toate că nici un calculator nu permite adresarea directă a biților, pentru compararea între calculatoare, dimensiunile din tabel se dau în biți.

TIP	DIMENSIUNE (biți)		ALINIERE (biți)	
	Calc.1	Calc.2	Calc.1	Calc.2
char	8	8	8	64
short	16	24	16	64
int	32	48	32	64
long	32	64	32	64
float	32	64	32	64
double	64	128	32	64
pointer la char	32	30	32	64
alți pointer	32	24	32	64
structuri	≥8	≥64	32	64

Fig. Organizarea datelor utilizată în două compilatoare C

Memoria Calc.1 este organizată pe octeți. Deşi fiecare octet are o adresă, setul de instrucțiuni maşină cere ca întregii scurți să fie poziționați la adrese pare iar întregii la adrese multiplu de 4. Compilatorul va respecta aceste reguli chiar dacă trebuie să sară peste 1÷3 octeți. De exemplu, pentru un caracter urmat de un întreg scurt se vor aloca 4 octeți deşi este necesar doar unul.

În Calc.2 fiecare cuvânt constă din 64 de biți, iar pentru adresarea unui cuvânt sunt permişi 24 biți. Pentru biții individuali în interiorul unui cuvânt există 64 de posibilități de plasare ⇒ sunt necesari 6 biți suplimentari (2^6=64) pentru a distinge între ele. ⇒ Un pointer la un caracter, plasat oriunde în interiorul cuvântului, necesită 30 biți: 24 pentru a găsi cuvântul şi 6 pentru poziția caracterului în cuvânt, ceea ce ar fi destul de complicat de implementat.

Puternica orientare pe cuvinte a setului de instrucțiuni al Calc.2 determină compilatorul să aloce un cuvânt complet chiar şi în situațiile în care, pentru reprezentarea tuturor valorilor acelui tip, ar fi suficienți mai puțini biți. De exemplu, pentru un caracter sunt necesari 8 biți, din motive de aliniere se vor aloca 64 biți. În interiorul unui cuvânt biții pentru fiecare tip de bază sunt în poziții specificate (de exemplu la început sau la sfârşit). Alt exemplu: pentru un caracter urmat de un întreg scurt se vor aloca două cuvinte, cumulând 128 biți; pentru un caracter se utilizează efectiv 8 biți din primul cuvânt iar pentru întregul scurt se utilizează 24 biți din al doilea cuvânt.