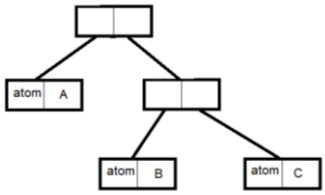
به نام خدا

شیرین حاجی امین شیرازی ۹۱۳۱۰۰۹ تمرین۴

۱- تمرین کتاب Mitchell

۳-۱-

(a



(b

$$\left(cons\ \left(cons\ 'A\ (cons\ 'B\ 'C)\right)(cons'B\ 'C)\right)$$

این دستور ، ابتدا لیست bc اول را می سازد ، سپس باز دیگر این لیست را ساخته و با a ، concat می کند و سپس کل دو لیست را به هم متصل می کند و لیست کلی را می سازد .

(c

$$((lambda\ (x)\ \left(cons\ (\ cons\ 'A\ x)(x)\right)\ \ (cons\ 'B\ 'C))$$

در این دستور لیست bc به روی apply x می شود . و در نتیجه لیست به اشتراک گذاشته می شود .

۳-۲-

a) خیر ، نمی توان این دستور را تفسیر کرد ، زیرا در صورتی که یکی از این k شرط ، مقداری نداشته باشند سیستم متوقف می شود (halt) و ادامه نمی یابد و نمی توان آن را شناسایی کرد.

b) برای پیاده سازی چنین سیستمی باید تمام شرط ها همزمان محاسبه شوند تا اگر یکی به جواب رسید و دیگری متوقف شد و یا تا ابد ادامه داشت سیستم به کلی متوقف نشود و یا محاسبه ادامه پیدا نکند اگر یکی از محاسبات هم زمان جواب داد ، مقدار آن انتخاب شود.

c) در این صورت شرط ها یا نباید اشتراک داشته باشند و یا اینکه هر دو شرطی که می توانند هم زمان درست باشند مقدار برابر برگردانند :

$$\left(define\ odd(x)\ cond\ \left((eq\ x\ 0)nil\right)\left((eq\ x\ 1)t\right)\left((>\ x\ 1\ )\left(odd(-x\ 2)\right)\right)\left((<\ x\ 0)\left(odd(+x\ 2)\right)\right)\right)$$

d) برای پیاده سازی scor ، روش اول بهتر است زیرا ابتدا e_1 محاسبه می شود و سپس به سراغ e_2 می رویم و e_1 نمی تواند تعریف نشده باشد زیرا سیستم متوقف می شود . اما برای Por ، روش دوم بهتر است زیرا محاسبه موازی صورت میگیرد و هرکدام true باشند ، مقدار مناسب محاسبه می شود.

-۴-۳

a) f(car xs)

b) g با maplist و H با car جایگزین می شود.

c) compose (f h)

-۵-۳

a) خیر زیرا ممکن است لیستی از جایی به بعد قابل دسترسی توسط برنامه ی ما نباشد اما دیگر برنامه ها همچنان به آن دسترسی داشته باشند . و یا اینکه همچنان پوینتر مربوط به آن وجود داشته باشد ولی در هیچ اجرایی از برنامه ما به آن دسترسی پیدا نکند.(و برنامه ای موجود باشد که بتواند به آن دسترسی داشته باشد و به این صورت توسط الگوریتم مک کارتی زباله تشخیص داده نشود.) و یا برای مثال توابع Inline که پس از تعریف و استفاده حتی اگر پوینتر آنها در حافظه باقی مانده باشد ، امکان فراخوانی و استفاده ی آنها وجود ندارد.

b) بله زیرا ادامه ی هر برنامه می تواند شامل هر ترتیبی از car و cdr ها باشد پس اگر توسط هیچ ترتیبی از اجرا قابل دسترسی نباشد ، توسط car و cdr رجیسترهای بیس نیز نیست.

c) امکان بررسی اینکه در تمام اجراهای ممکن برنامه هایی همزمان با هر ترتیبی چه خانه هایی از حافظه هرگز مورد استفاده قرار نمیگیرند قبل از اجرا بدون سربار وجود ندارد و در صورت امکان نیز سربار بسیار زیادی خواهد داشت. اما با استفاده از روش هایی مانند تولید گراف های حافظه بین برنامه ای می توان چنین collector هایی نوشت و علاوه بر آن گاهی به نظر می رسد لازم است state ای از متغیرها نگهداری شود. (مثلا شمارنده ای برای تعداد رفرنسها!)

-۶-۳

a)

$$cdr\big(cons\ (cons\ a\ b)(cons\ c\ d)\big): con(cons\ a\ b)(cons\ c\ d) \rightarrow count = 0(garbage\ cell)$$

$$,cons\ (a\ b) \rightarrow count = 0(garbage\ cell)$$

$$car\left(cdr\left(cons(cons\ a\ b)\big(cons\ (c\ d\ )\big)\right)\right): cons\ (c\ d) \rightarrow count = 0\ (garbage\ cell)$$

b) وقتی لیستی جایگزین car و یا cdr لیست دیگر می شود ، المنت جایگزین شده(خصوصا اگر لیست باشد) باید به حافظه آزاد برگردد . اما چون این تابع تغییری در پونتر ها ایجاد نکرده بود و سلول جدیدی ساخته نشده است ، این کار انجام نمی شود و به درستی حافظه عمل نمی شود.

-۸-۳

a)

$$t(n) = O(1) + \max\big(t(n-1), t(n-2)\big) = O(1) + t(n-1) \rightarrow t(n) \in O(n)$$

(b

$list$: (1 2 3)

$cons(rplcd\ list\ 4)\ (cdr\ list) \rightarrow$ (1 4 4)

$cons\ \big(future(rplcd\ list\ 4)\big)(cdr\ list) \rightarrow$ (1 4 2 3)

(c

در عبارت زیر همیشه v برگردانده می شود (بدون توجه به مقدار e) در حالی که در صورت عدم استفاده از future اینگونه نیست.

$$cond\big((future(e)p)(true\ v)\big)$$

d) در این صورت نمی توان به خطاها اولویت داد و اینکه اگر عبارت e شامل بیش از یک خطا باشد در اجراهای مختلف بر اساس ترتیبی که دستورات موازی انجام شده اند ، خطای تولید شده و نتیجه ی try متفاوت خواهد بود و امکان ردیابی آن وجود ندارد .

۲- گزارشی از garbage collection

[1]

نوعی مدیریت حافظهی خودکار است. این روش حالت خاصی از مدیریت منابع است که منابع محدود حافظه در آن که مدیریت می‌شوند و تلاشی برای بازستانی و بازیابی زباله یا حافظه‌ای که توسط اشیاء به کار گرفته شده و دیگر مورد نیاز برنامه نیست، خواهد بود . تکنیک زباله‌روبی توسط جان مک‌کارتی در حدود ۱۹۵۹ برای حل مشکلات لیسپ اختراع شده‌است.

زباله‌روبی اغلب در مقابل تنظیم دستی حافظه قرار دارد که با استفاده از آن ، برنامه‌نویس باید مشخص کند چه زمانی کدام قسمت های حافظه باید مورد استفاده قرار گیرند و کدام قسمت ها باز پس داده شوند و به حافظه اصلی برگردانند. اگر پروسه ای فضای مورد استفاده اش را رها سازی نکند ، برنامه تا انجا ادامه می یابد که یا کار پروسه به پایان برسد و یا کل حافظه اشغال شود و امکان فراهم سازی بقیه ی حافظه ی مورد استفاده ی برنامه وجود نداشته باشد . از طرفی اگر این کار به صورت دستی انجام پذیرد و به عهده ی برنامه نویس گذاشته شود ، ممکن است مشکلاتی از قبیل آزاد سازی زودتر از موعد حافظه، درخواست حافظه کمتر از میزان مورد احتیاج (memory leak) و یا عدم رهاسازی حافظه در زمان لازم و ... رخ دهد . بررسی ها نشان می دهدکه فرایند gc می تواند زمان اجرای برنامه را تا ۱۰ درصد زیاد کند .

فرایند gc را می توان به دو بخش اصلی زیر تقسیم نمود:

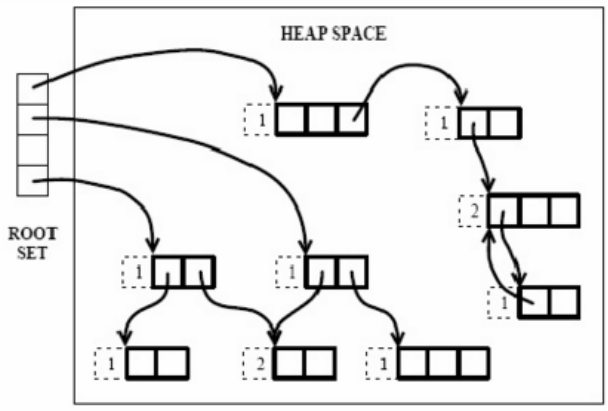
۱- تشخیص حافظه های زباله و تمییز آنها از حافظه ی مورد استفاده ی برنامه (garbage detection)
۲- بازگرداندن حافظه به حافظه ی آزاد قابل دسترسی (garbage reclamation)

برای این منظور روش های مختلفی مورد استفاده قرار می گیرد که در ادامه معرفی می شوند.

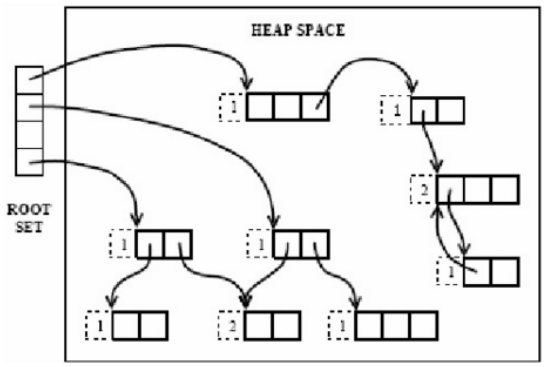
Reference counting

- هر object ، با یک شمارنده برای اشاره گر ها همراه می شود.

- هر بار که پوینتری به آن شی ساخته می شود، یکی به آن شمارنده اضافه شده و اگر پوینتری حذف شود ، شمارنده یک واحد کم می شود.
- اگر شمارنده به صفر برسد، حافظه ی مربو به آن شئ زباله در نظر گرفته شده و می تواند رها سازی شود .



- هرگاه حافظه ی مربوط به یک شئ بازیابی شود، تمام فیلد های آن بررسی شده و شمارنده ی تمامی اشیایی که به آنها اشاره می کرده ، یک واحد کم می شوند.
- این روش دو مشکل اصلی دارد :
  - نمی تواند در ساختار های حلقه ای به درستی عمل کند

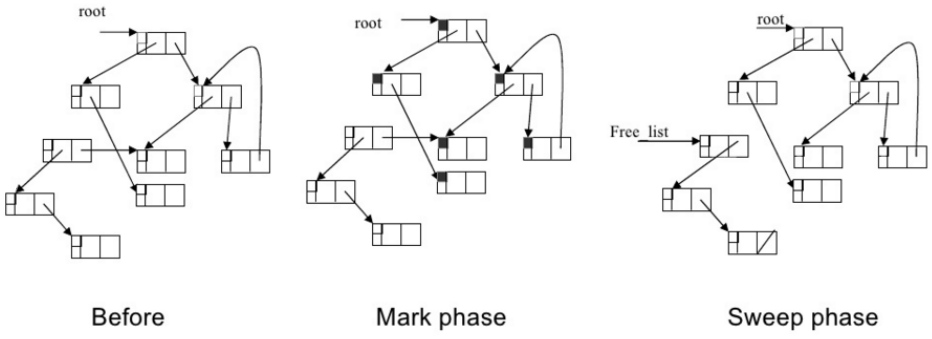


  - سربار زیادی برای نتظیم شمارنده ها بر سیستم پیاده می شود.

در مجموع می توان گفت این روش برای سیستم های high-performance مفید نیست ولی در بسیاری از سیستم ها که از ساختار های acyclic و بدون حلقه استفاده می کنند به کار گرفته می شود زیرا پیاده سازی ساده و قابل فهمی دارد .

Mark-sweep collection

- با شروع از ریشه و پیمایش گراف وابستگی اشاره گرها حافظه هایی که پیمایش می شوند علاتم زده می شوند (تگ ۱) و پس از اتمام پیمایش تمام مکان هایی از حافظه که علامت زده نشده باشند ، رهاسازی می شوند .



- این الگوریتم نیز مشکلاتی دارد :
  - ایجاد fragmentation در حافظه که باعث می شود تعریف شئ جدید حجیم سخت شود.
  - هزینه آن وابسته به اندازه Heap است
  - فرایند sweep به صورت تنبل انجام می شود و حافظه ی رها شده جمع آوری نمی شود .

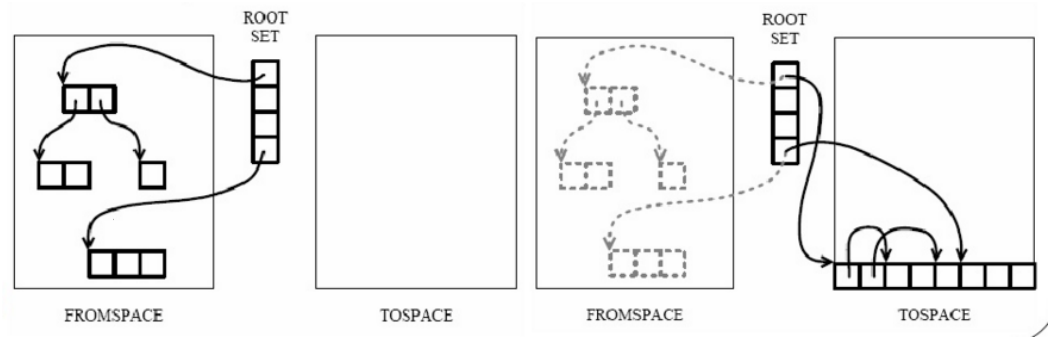
Mark-compact collection

در این روش حافظه های خالی در انتهای مموری جمع آوری می شوند .

- این روش نیازمند گذرهای بسیاری بر حافظه است. یکی برای تشخیص مکان جدید اشیا و به تبع آن چندین گذر برای تغییر اشاره گر ها و جابه جا کردن شئ .
- این الگوریتم در مقایسه با روش قبل می تواند بسیار آهسته تر باشد .
- این جمع آوری به رو روش two finger برای اشیای هم اندازه با دو گذر و الگوریتم لیسپ ۲ برای متغیرهای غیرهم اندازه وبا استفاده از یک اشاره گر اصافه برای هر شئ ، انجام می گیرد.

Copying garbage collection

- همانند الگوریتم mark-compact ، این الگوریتم تمام اشیای زنده را به یک نقطه از حافظه برده و در نتیجه باقی حافظه آزاد است و می تواند مورد استفاده قرار گیرد.
- در این روش حافظه به دو semispace تقسیم شده که در هر لحظه فقط یکی مورد استفاده است.
- این روش می تواند به صورت stop and copy انجام پذیرد که به صورت زیر عمل می کند:
  - حافظه به صورت خطی از قسمت مورد استفاده ی فعلی به برنامه ی درخواست کننده اختصاص داده می شود . اگر حافظه مورد درخواست بیش از فضای آزاد قسمت فعلی باشد، برنامه متوقف شده، gcصدا زده می شود .




```
initialize() =
  tospace = 0
  fromspace = N/2
  allocPtr = tospace

allocate(n) =
  If allocPtr + n > tospace + N/2
    collect()
  EndIf
  If allocPtr + n > tospace + N/2
    fail "insufficient memory"
  EndIf
  o = allocPtr
  allocPtr = allocPtr + n
  return o
```

```
collect() =
  swap( fromspace, tospace )
  allocPtr = tospace
  root = copy(root)

copy(o) =
  If o has no forwarding address
    o' = allocPtr
    allocPtr = allocPtr + size(o)
    copy the contents of o to o'
    forwarding-address(o) = o'
    ForEach reference r from o'
      r = copy(r)
    EndForEach
  EndIf
  return forwarding-address(o)
```

- در این روش حجم محاسبات در هر مجموعه وابسته به میزان داده ی زنده ی آن است.
- برای کم کردن تعداد بارهای به کار افتادن gc می توان semispace های بزرگتری را در ابتدا به برنامه اختصاص داد .
- اگر ram برای عملیات کپی کافی نباشد این روش قابل انجام نیست.

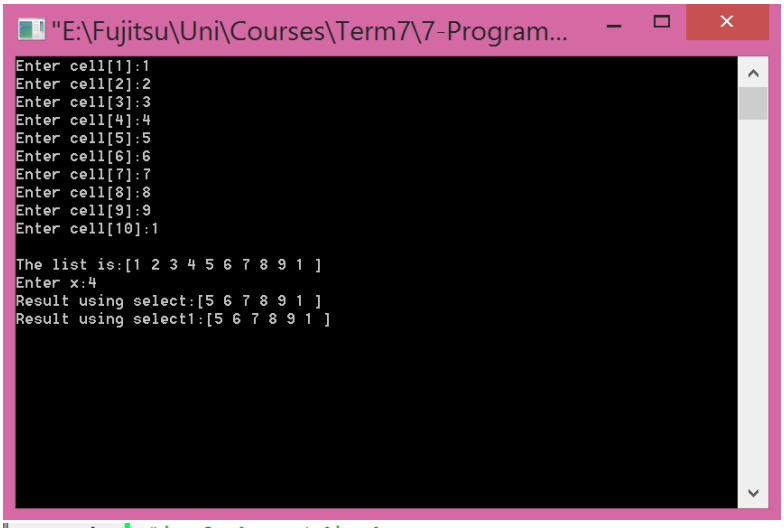
Non-copying implicit collection

- این روش به دو فیلد اشاره گر و یک فیلد رنگ اضافه برای هر شئ نیاز دارد .
- اشاره گر ها برای اتصال هر قسمت از حافظه به یک لینک لست دو طرفه به کار می روند.
- فیلد رنگ به آن منظور استفاده می شود که مشخص کند هر object متعلق به کدام دسته داده است .
- امکان compact کردن حافظه را ندارد .
- بر خلاف روش قبل برای trace کردن حافظه ی زنده و مورد استفاده نیاز به دو فضای to و from نیست . به همین منظور هزینه ی حافظه ی کمتری نسبت به روش قبل دارد .

[1] "Basic Garbage Collection Techniques." [Online]. Available: http://www.slideshare.net/khuonganpt/basic-garbage-collection-techniques. [Accessed: 20-Nov-2015].

۳- پیاده سازی تابع select در c

کد توابع معرفی شده در کلاس پیاده سازی شد . برای تست کد، فرض شد لیست فقط شامل اتم هاست و به دنبال اتم x در آن میگردیم . به منظور آزمون عملکرد لیستی از ۱۰ اتم ورودی گرفته می شود . به منظور راحتی فرض شد به جای سلول های اتم ، car سلول پدر ، type برابر atom دارد و atom_value مقدار آن را مشخص می کند . کد و نتایج به دست آمده در زیر مشاهده می شود.




```c
#include <stdio.h>
typedef struct cell cell;
struct cell{
    cell *cdr;
    enum { ATOM, NATOM } type;
    union {
            int atom_value ;
            cell* car_ptr;  // Units
    } car;
};
cell* select(cell*x ,cell*lst){
    cell* ptr;
    for(ptr=lst;ptr!=0;){
        if(ptr->car.atom_value== x)
            return ptr->cdr;
        else
            ptr=ptr->cdr;
    }
    return NULL;
}
cell* select1(cell*x ,cell*lst){
    cell* ptr , *previous;
    for(ptr=lst;ptr!=0;){
        if(ptr->car.atom_value == x)
            return ptr->cdr;
        else{
            previous=ptr;
            ptr=ptr->cdr;
            free(previous);
        }
    }
    return NULL;
}
```

```
34  main(){
35      cell* list= malloc(sizeof(cell));
36      int d;
37      printf("Enter cell[1]:");
38      scanf("%d",&d);
39      list->car.atom_value=d;
40      list->cdr=NULL;
41      cell*head=list;
42      int i=0;
43      for( i=0;i<9;i++){
44          printf("Enter cell[%d]:",i+2);
45          scanf("%d",&d);
46          cell* new_cell= malloc(sizeof(cell));
47          new_cell->car.atom_value=d;
48          new_cell->cdr=NULL;
49          head->cdr=new_cell;
50          head=new_cell;
51      }
52
53      head=list;
54      printf("\nThe list is:[");
55      for( i=0;i<10;i++){
56          printf("%d ",head->car.atom_value );
57          head=head->cdr;
58      }
59      printf("]\nEnter x:");
60      scanf("%d",&d);
61      head=select(d,list);
62      printf("Result using select:[");
63      if(head==NULL)
64          printf("X is either not in the list or the last element.");
65      while(head!=NULL){
66          printf("%d ",head->car.atom_value );
67          head=head->cdr;
68      }
69      printf("]\n");
70      head=select1(d,list);
71      printf("Result using select1:[");
72      if(head==NULL)
73          printf("X is either not in the list or the last element.");
74      while(head!=NULL){
75          printf("%d ",head->car );
76          head=head->cdr;
77      }
78      printf("]\n");
79      getch();
80
81
82  }
83
```