تبدیل الگوریتم‌های بازگشتی به غیربازگشتی:

الگوریتم‌های بازگشتی:

در ریاضیات کاربردی و به خصوص کامپیوتر مسائل فراوانی وجود دارد که حل آنها را به سادگی می‌توان به صورت یک الگوریتم بازگشتی نشان داد. یک الگوریتم بازگشتی مانند یک تابع و یا یک دنباله بازگشتی تعریف می‌شود فرمان‌های الگوریتم به طور مکرر و با پارامترهای مختلف اجرا می‌شوند تا به فرمان بنیادی الگوریتم برسیم. آنگاه تمام مقادیری را که محاسبه‌ی آن‌ها انجام نشده است را به صورت بازگشتی محاسبه می‌نماییم تا فرمان مورد نظر اجرا شود. یک روش متداول برای آسان سازی مسائل این است که آن‌ها را به زیر مسائلی از همان نوع تقسیم‌بندی کنیم. این روش با نام گویشی کردن شناخته می‌شود. به عنوان یک تکنیک برنامه‌نویسی کامپیوتر به این روش divide and conquer اطلاق می‌شود و کلید راه حل تعداد زیادی از مسائل کامپیوتری مهم است و یک بخش اساسی می‌باشد.

تمام زبان‌های برنامه نویسی که امروز مورد استفاده‌اند تعریف مستقیم از توابع بازگشتی را در خود دارند.

اکثر توابع و روش‌هایی که می‌توانند به وسیله‌ی کامپیوتر ارزشیابی شوند بدون استفاده از غیربازگشتی کردن قابل بازگشتی شدن هستند.

بازگشت مستقیم و غیرمستقیم:

بازگشت مستقیم زمانی است که تابع خود را فراخوانی کند و غیرمستقیم زمانی است که به طور مثال تابع الف تابع ب و تابع ب، تابع ث و تابع ث نیز دوباره تابع الف را فراخوانی کند.

بازگشتی در مقایسه با غیربازگشتی:

برای اینکه بازگشتی موفق باشد مسئله نیاز است که زیر ساختار بازگشتی داشته باشد. راه‌حل بعضی از مسائل به طور ذاتی بازگشتی است چون احتیاج به نگهداری حالت قبلی دارند. الگوریتم پیمایش درخت (tree traversal)، تابع اکرمن (Ackermann) و الگوریتم‌های تقسیم و غلبه مانند مرتب‌سازی سریع

(quick sort) همگی به صورت بازگشتی هستند. همه‌ی این الگوریتم‌ها می‌توانند به صورت غیر بازگشتی با کمک پشته هم پیاده شوند اما نیاز به پشته مزیت راه حل غیربازگشتی را از بین می‌برد.

تابع غیربازگشتی احتمالاً در عمل کمی سریع‌تر از نسخه بازگشتی آن اجرا می‌شود چون تابع غیربازگشتی سربار فراخوانی تابع (function-call) را به اندازه تابع بازگشتی ندارد و این سربار در بعضی زبان‌ها نسبتاً بالا است.

یک دلیل دگیر به ترجیح غیربازگشتی به بازگشتی این است که فضای پشته قابل دسترس کمتر از فضای قابل دسترس در حافظه آزاد heap است و الگوریتم‌های بازگشتی تمایل به فضای پشته بیشتری نسبت به غیربازگشتی دارند.

در ادامه به چگونگی تبدیل توابع بازگشتی به غیربازگشتی خواهیم پرداخت.

کاربرد پشته در تبدیل الگوریتم بازگشتی به غیربازگشتی:

شبیه سازی فراخوان‌های بازگشتی یکی دیگر از کاربردهای پشته است. این بحث عمدتاً برای فهم بهتر از رفتار الگوریتم‌های بازگشتی ارائه می‌شود و از آن می‌توان به عنوان شروعی برای تبدیل خودکار رویه‌های بازگشتی به رویه‌های غیربازگشتی معادل استفاده کرد.

هر فراخوانی شامل دو مرحله اصلی هستند:

1. عمل فراخوانی (call) که شامل مراحل زیر است:
   - ذخیره متغیر های محلی در پشته
   - آدرس بازگشت به پشته (return address)؛
   - انتقال پارامتر ها (parameter passing)
   - ذخیره PC در پشته؛
   - خروجی فراخوانی (return value).
2. بازگشت از یک فراخوانی (return) که شامل مراحل زیر است:
   - متغیر های محلی از پشته pop می شوند؛

- آدرس بازگشت از پشته pop می شود؛
- آخرین رکورد از پشته pop می شود(PC)؛
- ادامه کار از آدرس بازگشت.

نکته: Recursive و Non-Recursive از دید کامپایلر فرق ندارند

Call stack:

اکثر کامپایلرها برای فراخوانی و برگشت از زیربرنامه، call stack را پیاده‌سازی می‌کنند call stack یا run-time stack یک پشته است که اطلاعاتی درباره زیربرنامه فعال یک برنامه را نگهداری می‌کند. زیر برنامه فعال زیر برنامه‌ای است که فراخوانی شده است اما هنوز اجرایش تمام نشده است. وقتی زیربرنامه‌ای فراخوانی می‌شود، قبل از کنترل اجرای برنامه به آدرس زیربرنامه پرش کند آدرس دستورالعمل بعدی (دستورالعملی که در حافظه بعد از دستور فراخوانی قرار دارد) در جایی باید ذخیره شود که هنگام برگشت از زیر برنامه از آن استفاده می‌شود. این آدرس را آدرس برگشتی (return addresses) می‌نامند.

معماری که براساس پشته است آدرس برگشتی به عنوان نقطه برگشت در پشته اضافه می‌شود هر بار که زیر برنامه‌ای فراخوانی می‌شود آدرس برگشتی در پشته push می‌شود. هنگام برگشت از زیر برنامه آدرس برگشتی از پشته pop شده و کنترل برنامه به آن آدرس پرش می‌کند و اجرای برنامه بعد از دستور فراخوانی ادامه پیدا می‌کند.

به دلیل استفاده از پشته یک زیر برنامه می‌تواند خودش یا زیر برنامه‌های دیگر را صدا بزند.

عملکردهای call stack:

هدف اصلی یک call stack نگه داشتن آدرس برگشتی هر زیر برنامه فعال است اما بسته به زبان، سیستم عامل و محیط سخت افزاری ممکن است عملکردهای اضافی دیگری هم داشته باشد نظیر:

ذخیره آدرس‌های برگشتی:

برای هر برنامه یک پشته در نظر گرفته می‌شود. وقتی زیر برنامه‌ای در آدرس فراخوانی می‌شود آدرس دستورالعمل بعد از عبارت فراخوانی (آدرس برگشتی) در پشته قرار می‌گیرد. زیر برنامه می‌تواند به صورت بازگشتی باشد هر بار که زیر برنامه خودش را صدا می‌زند آدرس برگشتی در پشته ذخیره می‌شود.

ذخیره متغیرهای محلی:

متغیرهایی که درون زیر برنامه تعریف می‌شوند متغیرهای محلی نامیده می‌شوند. متغیرهای محلی تنها درمان زیر برنامه فعال شناخته شده هستند و بعد از تمام زیر برنامه مقدار آن‌ها در حافظه باقی نمی‌ماند. اغلب مناسب که فضایی در پشته به آن‌ها اختصاص داده شود که سریع‌تر از تخصیص فضای heap به آن‌ها است. هر زیر برنامه فعال فضای جداگانه خودش را در پشته برای داده‌های محلی دارد.

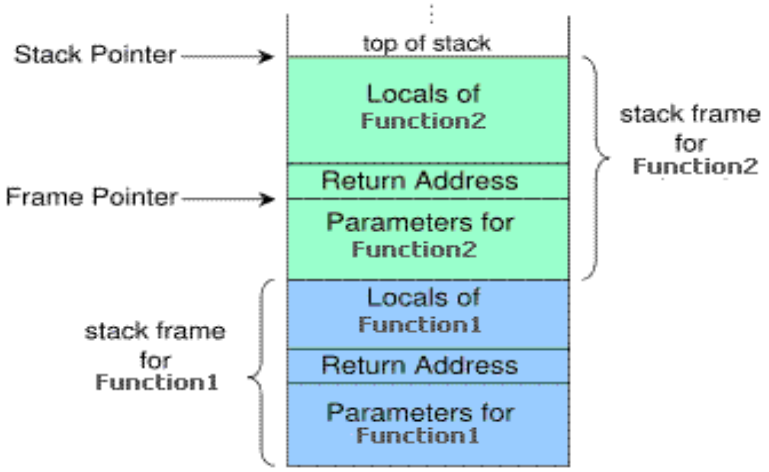
ارسال پارامتر

مقادیر پارامترهای مورد نیاز زیر برنامه‌ها هنگام فراخوانی به آن‌ها داده می‌شوند. معمولاً فضایی از call stack برای ذخیره مقدار این پارامترها اختصاص داده می‌شود. هر فراخوانی به زیر برنامه مقادیر مختلفی از پارامترها را خواهد داشت و فضای جداگانه‌ای در پشته به آن‌ها داده می‌شود.

ساختار call stack:

یک call stack از stack frameها یا activationها تشکیل شده است. فریم پشت اطلاعات زیر برنامه را نگه می‌دارد. هر فریم پشته مربوط به یک فراخوانی زیر برنامه‌ای است که هنوز تمام نشده است.

مثال: فرض کنید تابع function2 اکنون در حال اجراست و توسط function1 فراخوانی شده است وضعیت پشته می‌تواند به شکل زیر باشد.



فریمی که در بالای پشته است مربوط به زیر برنامه‌ای است که اکنون در حال اجراست.

هر فریم ممکن است دربرگیرنده متغیرهای محلی، آدرس برگشتی و مقدار پارامترهای زیر برنامه باشد

فریم‌های پشته هم اندازه نبوده و زیر برنامه‌های مختلف فریم‌های متفاوتی دارند.

پشته توسط stack pointer دسترسی می‌شود که بالای پشته را مشخص می‌کند.

برج‌های هانوی:

حال می‌خواهیم با ذکر مثال برج‌های هاندی به جزئیات دقیق‌تر این بخش اشاره کنیم.

HONOI (n,f,t,h)

سکه را از میله‌ی f به میله‌ی t منتقل می‌کند.

1. if n=1

2. Then print f ⊃ t سکه بالایی را از میله f به میله t منتقل کن

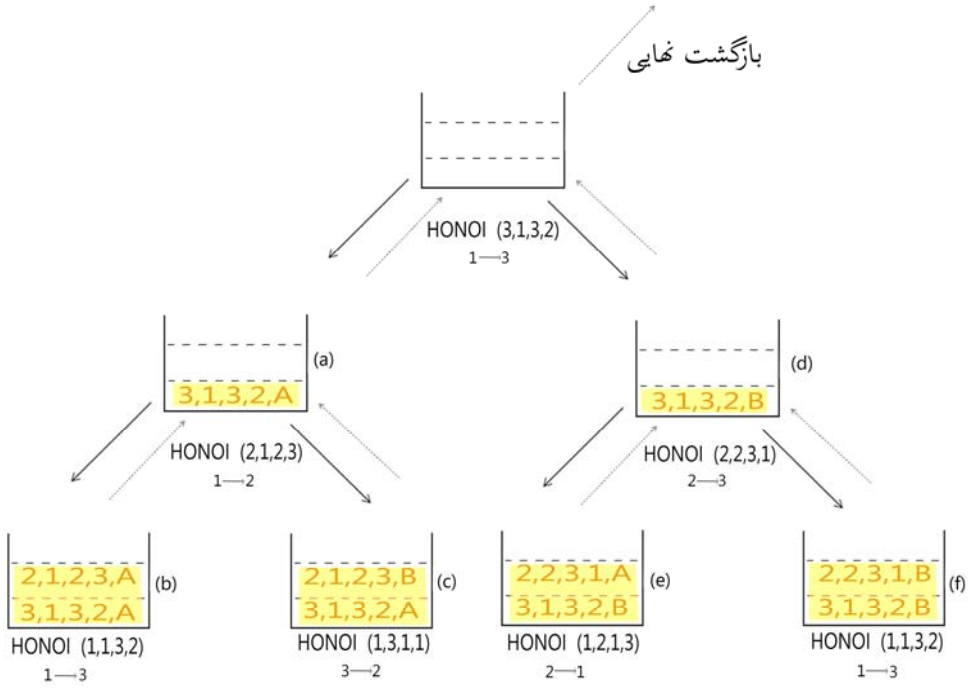
3.else HONOI (n-1 f, h, t)

4. A: print f ⊃ t سکه بالایی را از میله f به میله t منتقل کن

5. honot (n-1, h,t,f)

6. B:

برای درک بهتر، مراحل مختلف فراخوانی و بازگشت برای اجرای HONOI(3,1,3,2) در شکل زیر نشان داده شده است.



در ابتدا پشته خالی است و اولین فراخوانی انجام می‌شود. مطابق الگوریتم، این موجب فراخوانی بازگشتی HONOI(2,1,2,3) می‌شود. برای شبیه سازی، نخست مقادیر فعلی متغیرهای h,t,f,n و سپس آدرس بازگشت در بالای پشته قرار می‌گیرند. در این حالت f=1

n=3، t=3 و h=2 و آدرس بازگشت را A فرض می‌کنیم که در رویه‌ی HONOI به عنوان آدرس 3 نشان داده شده است. بنابراین رکورد بالای پشته (3,1,3,2,A) خواهد بود. انتقال پارامتر انجام می‌شود، یعنی نگارش‌های n=2 و f=1 و t=2 و h=3 اجرا و کار الگوریتم از سطر اول آن دنبال می‌شود. این با حالت (a) در شکل نشان داده شده است. HONOI(2,1,2,3) هم مانند یک فراخوانی مستقل اجرا می‌گردد که خود موجب فراخوانی HONOI(1,1,3,2) می‌گردد، ولی پیش از آن، رکورد (2,1,2,3,A) در بالای پشته قرار می‌گیرد. (حالت b). چون n=1، این فراخوانی دستور شماره 1 را اجرا می‌کند و در خروجی 3 ⊐ 1 را می‌نویسد تا بالاترین ملکه از میله 1 به میله 3 منتقل شود.

سپس عمل بازگشت انجام می‌شود؛ به این صورت که مقادیر بالای پشته به متغیرها گمارده می‌شود (یعنی مجدداً به حالت (a) برمی‌گردیم) و کار از دستور A با این مقادیر دنبال می‌شود. یعنی2 ⊐ 1 در خروجی نوشته می‌شود. (چون f=1 و t=2 است) و HONOI(1,3,2,2) فراخوانده می‌شود. توجه کنید که آدرس

بازگشت این فراخوانی B است و این مقادیر در بالای پشته درج می‌شود. این فراخوانی 2⊐ 1 را چاپ می‌کند و پس از POPمقادیر بالای پشته و گمارش به متغیرها، کار را از آدرس B، که خود یک بازگشت است، دنبال می‌کند. بنابراین پس از رفتن به حالت (a) از حالت (c)، مجدداً مقادیر بالای پشته pop شده (یعنی به حالت اول با پشته خالی باز می‌گردیم) و کار از آدرس A دنبال می‌شود.

سپس به حالت پشته‌ها به ترتیب به (d)، (e) و (f) می‌رود و با سه بار بازگشت در پایان به برنامه‌ای باز می‌گردد که این رویه را فراخوانده است. اگر ترتیب حرکت‌ها را دنبال کنید می‌بینید که حرکت‌ها به درستی انجام شده‌اند.

حال می‌خواهیم یک رویه‌ی غیربازگشتی بنویسیم که همین مراحل کار رویه‌ی بازگشتی را که گفتیم شبیه‌سازی کند. برای این کار یک پشته‌ی s را که در ابتدا تهی است ایجاد می‌کنیم. هر رکورد این پشته شامل 4 مقدار عددی، برای متغیرهای n، f، t و h و یک مقدار نویسه‌ای A یا B به عنوان آدرس بازگشت است. سطر با برچسب Rec-call در این رویه به معنی شروع یک ساختمان یک فراخوانی بازگشتی است که با Rec-call آغاز می‌شود. سطر یا برچسب return-label هم شروع عمل بازگشت است.

```
NONRECURSIVE-HONOI (n,f,t)
1. پشته s شامل آدرس بازگشت و مقادیر همه‌ی متغیرهای محلی است
2. CREATE-STACK(S)
3. h ⊏ the other peg
4. دستور شماره 4 آغاز یک فراخوانی بازگشتی است
5. Rec-Call: if n=1
6.      then print f ⊐ t  سکه بالایی را از میله f به میله t منتقل کن
7.           goto Return-Lable
8.      else push (S,Stack Rec (n,f,t,h,a)
9.           n,f,t,h ⊐ n-1,f,h,t  انتقال پرامترها با فرض ارزشی بودن
10.          goto Rec-Call
```

از این دستور عمل بازگشت شبیه سازی می‌شود

```
11. Return-Label: if not Is EMPTY(S)
12.       then      return-address, n,f,t,h ⊲ pop(s)
13.                 switch
14.                       case return-address=A
15.             do print f ⊳ t   سکه بالایی را از میله f به میله t منتقل کن
16.                       push (s,STACKRec (n,f,t,h,B)
                          انتقال پارامترها
17.                       n,f,t,h  ⊳  n-1, h,t,f
18.                       goto Rec-Call
19.                       case return-address=B
20.                       do goto Return=Lable
```

مانند الگوریتم بازگشتی، هر فراخوانی در این الگوریتم از سطر 4 آغاز می‌شود. اگر n=1 باشد، تنها سکه (در عمل سکه رویی) را حرکت می‌دهد و باز می‌گردد، وگرنه مقادیر فعلی متغیرها و آدرس بازگشت A را در بالای پشته درج می‌کند و پس از انتقال پارامترها برای انجام یک فراخوانی بازگشتی دیگر به سطر می‌رود. سطر 11 که عمل بازگشت را شبیه سازی می‌کند: اگر پشته تهی باشد، یعنی این آخرین بازگشت است و باید به برنامه‌ای که این رویه را فراخوانده است باز گردد. اگر پشته تهی نبود، باید مقدارهای بالای پشته پس از دریافت و گمارش به متغیرها دور ریخته شود و بسته به آدرس بازگشت، دو کار مختلف انجام دهد: یکی انجام یک فراخوانی بازگشتی دیگر (با آدرس بازگشت B) و دیگری که خود یک بازگشت است با یک goto در سطر 18 انجام می‌شود. آنچه گفته شد را می‌توان به صورت خودکار، برای رویه‌های بازگشتی با پارامترهای ارزشی هم پیاده سازی کرد. اگر پارامتری آدرسی باشد، باید به جای مقدار آن آدرس آن را در پشته ذخیره کرد.

مثال‌ها:

حال که به بررسی چگونگی تبدیل الگوریتم‌های بازگشتی به غیربازگشتی اشاره کردیم می‌خواهیم مثال‌هایی دیگر را در این زمینه بیان کنیم.

مثال 1: تابع بازگشتی و غیربازگشتی محاسبه‌ی فاکتوریل یک عدد صحیح را بنویسید.

تابع بازگشتی

```
1. int Factorial (int x)
2. {
3.     if (x <= 1)
4.         Return 1;
5.     else
6.         return n*Factorial (x-1);
7. }
```

////////

تابع غیربازگشتی

```
1. int Factorial (int x)
2. {
3.     int I, temp;
4.     for (j=1; j<= x; j++)
5.     temp*=j;
6.     Return temp;
7. }
```

مثال 2: تابع بازگشتی و غیربازگشتی الگوریتم اقلیدسی برای محاسبه‌ی بزرگترین مقسوم علیه مشترک دو عدد صحیح را بنویسید.

تابع بازگشتی

```
1. int GCD (int x, int y)
2. {
```

```
3.      if (y==0)
4.              return x;
5.      else
6.              return GCD (y, x%y);
7. }
```

////////

تابع غیربازگشتی (که نیاز به یک متغیر موقت دارد)

```
1. int GCD (int x, int y)
2. {
3. while (y!=0){
4.      int r= x%y;
5.      x=y;
6.      y=x;
7.      return x;
8. }
```

مثال 3. تابع بازگشتی دنباله فیبوناچی را بنویسید.

```
1. Function Fibonacci (n: Integer): Integer;
2. Begin
3. if (n=1) or (n=2) Then Fib:=1
4. Else Fibonacci:= Fibonacci (n-2)+Fibonacci (n-1);
5. end;
```

مثال 4: تابع بازگشتی اکرمن (Ackermann function) را بنویسید.

```
1. Function Ackermann (a,b: Integer): Integer;
2. Begin
```

3. If $(a < 0)$ and $(b < 0)$ Then Ackermann: = 0

```
4. Else if a=0 then Ackermann: = b+1
```

5. Else if b=0 Then Ackermann: = (b-1,1)

6. Else Ackerman: = Ackerman (a-1, Ack (a,b-1));

7. End;

مثال 5: تابع بازگشتی comb زیر را به یک تابع غیر بازگشتی تبدیل کنید.

```
void RecComb(int m, int n, int &res)
{
1.      int res1, res2;
2.      if(m==n or m==0)
3.      {
4.              res = 1;
5.              return;
6.      }
7.      RecComb(m, n-1, res1);
8.      RecComb(m-1, n-1, res2);
9.      res = res1 +res2;
10.  }
```

حل:

```
int NonRecComb(int m, int n)
{
1.      STACK s;
2.      int res1, res2, res;
3.      Char addr;
```

```
4.  START:
5.      if(m==n or m==0)
6.      {
7.          res = 1;
8.          goto RET;
9.      }
10.     s.push(m, n, res, res1, res2, 'A');
11.     n--;                          //Parameter passing
12.     goto START;
13. RET:
14.     if(s.isEmpty())
15.         return res;
16.     int temp = res;
17.     s.pop(m, n, re1, res2, res, addr);
18.     if(addr == 'A')
19.     {
20.         res1 = temp;
21.         s.push(m, n, res, res1, res2, 'A');
22.         n--;
23.         m--;
24.         goto START;
25.     }else{
26.         res2 = temp;
```

```
27.                 res = res1 + res2;
28.                 goto RET;
29.         }
```