EOPL Pset 5

Alireza Habibzadeh `99109393`

1

خروجی این برنامه 0 است چرا که با هر بار اجرای این تابع مقدار `counter` برابر با 0 می‌شود و سپس یکی زیاد می‌شود. سپس مقدار 1 خروجی داده می‌شود. در هر دو اجرای تابع همین اتفاق می‌افتد پس خروجی هر دو 1 است و تفاضلشان 0.

برای حل ایراد کد (اگر قرار است تابع به عنوان شمارنده استفاده شود) می‌توان آدرسی که در `counter` نگه داشته می‌شود (که عدد شمارنده ما در آن ذخیره می‌شود) را تنها یکبار و قبل از تعریف تابع تعیین کرد.

```
let counter = newref(0)
  in let g = proc (dummy)
    in begin
      setref(counter, -(deref(counter), -1));
      deref(counter)
    end
  in let a = (g 20)
    in let b = (g 20)
      in -(a, b)
```

پس از این اصلاح با هر بار اجرای تابع مقداری که `counter` در حافظه به آن اشاره می‌کند یکی زیاد می‌شود پس حاصل برابر با

$$a - b = 1 - 2 = -1$$

است.

2

برای این کار چند روش به ذهن من می‌رسد. یکی از این‌ها وقتی قابل اجرا است که زبان ما امکان کار با referenceها چه به صورت implicit و چه به صورت explicit داشته باشد. در این صورت هر ورودی تابع که به صورت یک reference باشد عملا خروجی هم هست! چرا که درست است که نمی‌توانیم خود مقدار متغییر در جایی که تابع ما را call کرده تغییر دهیم (عملا خود آدرسی که به ما پاس داده شده) اما می‌توانیم مقداری که در آن‌جای حافظه ذخیره شده را تغییر دهیم. مثال زیر به زبان IMPLICIT-REFS نوشته شده است:

```
let rotate = proc (a, b) in
  let temp = -(a, 0) in ; to make temp a deep copy of a
    begin
    set a = -(0, b);
    set b = temp;
    34
  end
  in let x = 20
    in let y = 30
      in (rotate 20 30)
```

$$(x, y) \to (-y, x)$$

خروجی برگشتی تابع یک مقدار dummy برابر با 34 است. اما خروجی واقعی تابع تغییر کردن متغییرهای `x` و `y` پاس داده شده به آن هستند.

یک روش دیگر برای خروجی دادن چند متغییر خروجی دادن یک لیست است. در زبان `let-rec` که به کلی از referenceها هم پشتیبانی نمی‌کند می‌توان خروجی‌های تابع را در یک لیست ریخت و آن لیست را خروجی داد و از مقادیر جداگانه استفاده کرد. البته باید دقت کنیم برای کامل بودن این روش بهتر است بتوان اعضای لیست از جنس‌های مختلف باشند. (برخلاف بعضی زبان‌ها و بعضی ساختارهایشان که همه‌ی اعضای لیست باید از یک جنس باشند)

روش دیگر استفاده از structها و ساختارهای داده است که هم در زبان racket و هم در بسیاری از زبان‌ها پشتیبانی می‌شود. می‌توانیم برای هر تابعی که نیاز به خروجی‌های متعدد دارد یک struct بسازیم که آن را به عنوان خروجی تحویل دهد.

در انتها روشی که در کلاس هم اشاره شد استفاده از mutable-pairها است. این‌ها یک ساختار جدید هستند که اجازه می‌دهند دو مقدار به صورت mutable در یک ساختار واحد کنار هم قرار گیرند. البته این روش‌ها شبیه هم هستند و به نوعی این روش حالت خاص روش قبلی است.

3

ابتدا به این نکته توجه می‌کنیم که در یک زبان pure functional مقدار هر متغییر همواره یک چیز است و side effect حافظه‌ای نداریم. پس عملا اثبات شد که برای این زبان‌ها هر دو روش call-by-need و call-by-name نمی‌تواند فرقی داشته باشد چرا در هر دو روش مقدار متغییر چه یکبار ارزیابی شود و چه چند بار همواره یکسان است.

پس برای ایجاد تفاوت باید یک side effect حافظه با ارزیابی شدن متغییر باشیم. از شمارنده‌ی سوال 1 تمرین استفاده می‌کنیم.

```
let counter = newref(0)
  in let g = proc (dummy)
    in begin
      setref(counter, -(deref(counter), -1));
      deref(counter)
    end
  in let a = (g 20)
      in -(a, a)
```

در زبانی که lazy و call-by-need باشد مقدار `a` در expression آخر تنها یکبار ارزیابی شده و حاصل تفریق برابر با 0 است اما در زبان‌های lazy و call-by-name مقدار `a` در expression آخر دو بار ارزیابی می‌شود که در بار اول برابر با 1 و بار دوم برابر با 2 است پس حاصل برابر با 1- و متفاوت با زبان دیگر شد.

4

4.1

> پیاده‌سازی زبان آقای پاسکال خیلی ساده بود. call-by-value و call-by-reference دقیقا همون مفاهیمی بودن که تو زبان پیاده‌سازی وجود داشتن.

پیاده‌ساز زبان

> پاس دادن رفرنس و مقدار دیگه چیه؟ این‌ها از مفاهیمی که من می‌فهمم خیلی دورن. وقتی لیست‌های حجیم رو با مقدار به تابع پاس می‌دم برنامه کند می‌شه. وقتی هم چیزی رو با رفرنس پاس میدم مطمئن نیستم که تابع مقدار داخل اون‌ها رو تغییر میده یا نه.

برنامه‌نویس

4.2

به جز تعریف روتین و کلیشه‌ای چیز دیگری به ذهنم نرسید، زبان پاسکال سطح پایین‌تر است چرا که از مفاهیمی استفاده می‌کند که به مفاهیم پیاده‌سازی شده در کامپایلر، ماشین و کدهای زیرین نزدیک‌تر است اما چیزی که در زبان خانم ادا استفاده شده از آن چه به طور معمول در کامپایلرها، کدهای زیرین و کد ماشین استفاده می‌شود دورتر است و نیاز است تا مفهوم زبان خانم ادا به طور مفصل در مفسر یا کامپایلر زبان پیاده‌سازی شود، گویی زبان خانم ادا *روی* این پیاده‌سازی مفصل نشسته بنابراین *سطح بالاتر* است.

حال اگر ماشینی داشته باشیم که کد خانم ادا برای آن مثل ماشین-کد عمل کند اما کد پاسکال را باید با زبان خانم ادا تفسیر کرد آیا باز هم باید گفت زبان خانم ادا سطح بالاتر است؟ به نظر من در این مورد خاص خیر اما منظور ما از سطح بالایی و پایینی با توجه به معماری‌های مرسوم است که این معماری‌ها هم برخاسته از سیر معمول مفاهیم ساده (مثل متغییر و حافظه و آرایه و عملیات‌های جبری) به مفاهیم پیچیده هستند. مسلما ما در مورد سخت‌افزارهایی صحبت نمی‌کنیم که ماشین‌کد آن از توابع مثلثاتی پشتیبانی می‌کند اما از توابع جبری ساده خیر. هرچند ساده‌ترین تعریفی که از این مفاهیم ریاضی داریم همین معماری مرسوم را پیشنهاد می‌دهند، برای پیاده‌سازی توابع مثلثاتی در سطح سخت‌افزار احتمال زیاد نیاز به پیاده‌سازی توابع جبری در سطح سخت‌افزار خواهید داشت که این یعنی سطح‌بالا بودن این توابع نسبت به همان توابع جبری به جای کد در سخت‌افزار پنهان شده است.

بحث شبیه به این بحث درباره‌ی پردازنده‌هایی است که مستقیما بایت‌کد Java اجرا می‌کنند. در این پردازنده‌ها زبان C سطح بالاتری دارد یا بایت‌کد Java؟

4.3

ساده‌ترین و طبیعی‌ترین روش پیاده‌سازی به نظر من این است که برای `proc-in` از فراخوانی با مقدار و برای `proc-inout` از فراخوانی با رفرنس استفاده شود برای `proc-out` هم از فراخوانی با رفرنسی استفاده شود که dereference کردن آدرس آن در تابع ممنوع شود و تنها می‌توان مقداری که آدرس به آن اشاره می‌کند را set کرد.

این روش تضمین می‌کند که بدون کار اضافه‌ای ویژگی‌های مورد نظر خانم ادا پیاده‌شود. اما مشکل این روش این است که برای مقادیر `proc-in` ای که حجم زیادی دارند غیر بهینه عمل می‌کنیم. اگر آن‌ها را هم به صورت رفرنس پاس دهیم اما در ابتدای کار چک کنیم که هیچ setای به آن‌ها صورت نگرفته باشد برنامه برای موارد خاص سریع‌تر کار می‌کند اما این چک خود می‌تواند زمان‌بر باشد.

4.4

ابتدا توابعی را در نظر می‌گیریم که تنها یک نوع از ورودی، خروجی یا ورودی/خروجی دارند. می‌دانیم بعدا با ترکیب این توابع (توابع مرتبه بالاتر) می‌توان توابعی با چند ورودی، خروجی یا ورودی/ خروجی ساخت.

پس با این اوصاف می‌توان گفت ما سه نوع تابع داریم، تابع `proc-in` تابع `proc-out` و تابع `proc-inout` حال باید تغییرات تابع `value-of` را بررسی کنیم. در این تابع سه case برای هر کدام از این توابع خواهیم داشت.

در حالت `proc-in` همان کار قبلی را می‌کنیم. یا تابع را بر روی مقدار deref شده‌ی متغییر اجرا می‌کنیم یا یک newref می‌سازیم و تابع را با متغییر جدید که کپی است اجرا می‌کنیم.

در حالت `proc-inout` مستقیما رفرنس متغییر که از environment بدست می‌آید را به تابع می‌دهیم. `(apply-env env var)`

در حالت `proc-out` به عنوان یک پیاده‌سازی ساده می‌توانیم از همان کد قبلی حالت `proc-inout` استفاده کنیم اما برای این که مطمئن شویم تابع از مقدار متغییر استفاده نمی‌کند آن را صفر کرده و سپس پاس می‌دهیم. `beign (set var = 0; (apply-env env var)`

البته یک مشکل این پیاده‌سازی این است که توابع `proc-out` حالت دست‌نزن ندارند. یعنی حالتی وجود ندارد که تابع بتواند اصلا خروجی‌ی در متغییر out خود ندهد و حتما باید خروجی دهد وگرنه خروجی پیشفرض که 0 است به out تابع assign می‌شود.

```
(define value-of
  (λ (exp env)
    (cases expression exp
      (const-exp (num) ...as before...)
      (diff-exp (exp1 exp2) ...as before...)
      (zero?-exp (exp1) ...as before...)
      (if-exp (exp1 exp2 exp3) ...as before...)
      (call-exp-in (rator rand)
                   (value-of (call-exp rator (- rand 0)) ρ))
      (call-exp-out (rator rand)
                    (begin
                      set rand = 0;
                      (call-exp rator rand)))
      (call-exp-inout (rator rand) (call-exp rator rand))
      ;
      ;
      ;
)))
```