

프로그래밍 언어 개론

과제 보고서

[PL00]HW3_201601980_KimSeongHan

개발환경 : Ubuntu

제출일 (2021-03-23)

201601980 / 김성한(00분반)

<과제 설명>

HomeWork1

Lists of pairs

```
val split : ('a * 'b) list -> 'a list * 'b list
  Transform a list of pairs into a pair of lists: split [(a1,b1); ...; (an,bn)] is
  ([a1; ...; an], [b1; ...; bn]). Not tail-recursive.

val combine : 'a list -> 'b list -> ('a * 'b) list
  Transform a pair of lists into a list of pairs: combine [a1; ...; an] [b1; ...; bn] is
  [(a1,b1); ...; (an,bn)].
  Raises Invalid_argument if the two lists have different lengths. Not tail-recursive.
```

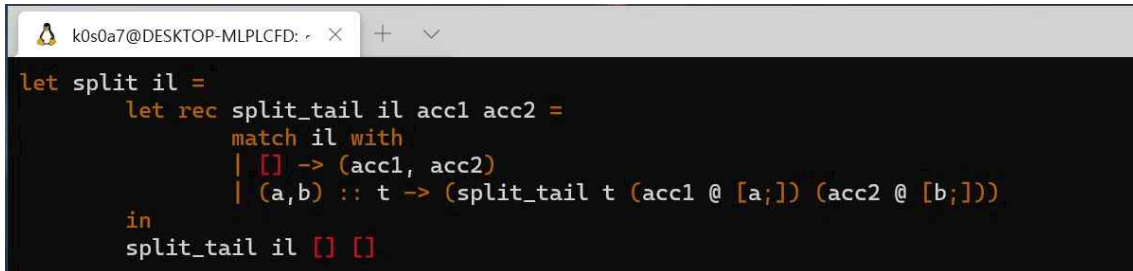
- Ocaml list 모듈에 구현된 함수를 tail-recursive하게 구현
- 두 list는 길이가 같다고 가정
- What is tail-recursion?

[문제 HomeWork1]

● 해당 과제는 기존에 ocaml에 정의되어있는 함수인 split 과 combine을 tail-recursive하게 구현하는 과제였다. 먼저 split함수의 타입은 ('a * 'b)list -> 'a list -> 'b list 로 원소 두 개로 이루어진 튜플들의 리스트를 인자로 받아, 두 개의 리스트로 분리하여 튜플의 형태로 반환하는 함수이다. combine 함수의 타입은 'a list -> 'b list -> ('a * 'b)list 로 두 개의 리스트를 인자로 받아 원소 두 개로 이루어진 튜플들의 리스트를 반환하는 함수이다.

● 처음 문제를 보고 pattern-matching을 사용하여 각각의 경우에 대해 리스트를 나누고 접합하는 연산이 필요함을 생각하면서 과제를 시작하였다.

<코드 구현 및 실행 결과 화면>



```
let split il =
  let rec split_tail il acc1 acc2 =
    match il with
    | [] -> (acc1, acc2)
    | (a,b) :: t -> (split_tail t (acc1 @ [a;]) (acc2 @ [b;]))
  in
  split_tail il [] []
```

[tailOpt.ml 파일 내부에 구현된 split 함수의 코드]

● 초기 `let rec split_tail il acc1 acc2` 으로 함수를 선언함으로써 `tail-recursive`하게 구현할 수 있도록 `acc`인자를 추가하였다. 그런데 생각을 하던중 두 개의 리스트를 마지막에 튜플의 형태로 반환해야하므로 `acc`인자가 두 개가 필요하다 생각하고 `acc1` 과 `acc2` 두 개로 선언해주었다.

● 그 후 `pattern-matching`을 사용하였다. 인자로 들어온 튜플들의 리스트 `il`에 대하여 재귀적으로 함수를 호출할 때, `il`이 비어있다면, `(acc1, acc2)`의 튜플을 리턴하고, 비어있지 않은 경우엔 튜플하나 `(a,b)`를 분리하여 `a`는 `acc1`리스트와 연결하고, `b`는 `acc2`와 연결해주었다. 그렇게 추가된 `acc`인자들과 `(a,b)`를 분리하고 남은 `t` 리스트를 다시 `split_tail`함수에 인자로 넘겨줌으로써 재귀문을 작성하였다. 이렇게 `acc`인자를 이용하여 `tail-recursive`하게 구현하였으며, 이 과정에서의 장점으로선 재귀문이 다시 연산을 하기위해 `call stack`에 쌓일 필요가 없어 `stack overflow`의 발생을 막을 수 있다.

● `split`의 내부에 들어갈 `split_tail`함수를 완성하고, 가장 바깥쪽을 `split`함수로 선언한 뒤 기존의 `split` 함수와 타입을 동일하게 맞춰주었다. 이 과정에서 `split` 내부에서는 `split_tail` 함수를 주어진 리스트인자 `il`로 호출하고 있으며, 초기 `acc` 의 인자들로는 비어있는 리스트(`[]`)를 넘김으로써 함수를 구현하였다. 래퍼함수 `split`함수의 인자와 내부 `split_tail`의 동작으로 미루어 볼 때 해당 함수의 타입은 `('a * 'b)list -> 'a list -> 'b list` 이 됨을 알 수 있다.

```

let combine il1 il2 =
  let rec combine_tail il1 il2 acc =
    match il1,il2 with
    | [],[] -> acc
    | h1::t1, h2::t2 -> combine_tail t1 t2 (acc @ [(h1,h2);])
    | ([],_::_) | (_::_,[]) -> failwith "Unsupported"
  in
    combine_tail il1 il2 []

```

[tailOpt.ml 파일 내부에 구현된 combine 함수의 코드]

● 초기 let rec combine_tail il1 il2 acc 로 combine_tail 함수를 선언하였다. combine 함수의 경우에도 기존 정의된 combine함수의 타입과 다르게 acc 인자를 추가해줌으로써 tail-recursive하게 구현할 준비를 해주었다.

● 이 경우에도 pattern-matching을 사용하였다. 초기 il1,il2에 대하여 두 리스트 모두 비어있다면 acc(원소들이 튜플의 형태로 저장되어 누적된 최종 리스트)를 반환하도록 해주었고, 두 리스트에 원소가 있는 경우엔 각각의 리스트에서 가장 앞에 원소를 하나씩 떼어내서 acc @ [(h1,h2);] 로 acc에 튜플의 형태로 원소를 추가하면서 combine_tail함수의 인자로 넘겨주었다. 이때, t1,t2(il1과 il2에서 원소를 하나씩 떼어내고 남은 리스트)를 함께 인자로 줘어주면서 combine_tail함수를 tail_recursive하게 구현하였다. 마지막으로 ([],_::_) | (_::_,[]) 인 경우인데 이 경우는 컴파일시에 에러항목에서 확인하면서 알게되었다. pattern-matching을 할 경우엔 모든 상황에 대한 고려가 필요한데 과제에서 두 개의 리스트의 길이가 같다고 주어져서 사실 신경을 쓰지 않고 있었지만 컴파일시에 에러로 나오는 것을 확인하고 추가하게 되었다. 해당 경우는 두 리스트의 길이가 다를 때 한 리스트만 비어있는 경우를 matching한 경우로 이번과제에선 해당사항이 없어 failwith으로 처리해주었다.

● split함수와 마찬가지로 combine함수도 기존에 정의된 combine 함수와의 타입을 맞춰주기 위하여 래퍼함수 combine으로 선언하였다. 기존 combine함수의 타입인 'a list -> 'b list -> ('a * 'b)list를 맞춰주기 위하여 래퍼함수의 인자로 두 개의 리스트를 받았고 내부에서 combine_tail함수를 let-in 구문으로 선언한뒤 acc인자에 빈 리스트를 주고 il1과 il2 인자로 호출해줌으로써 래퍼함수를 통해 함수의 타입이 바뀌지 않도록 구현하였다.

```
k0s0a7@DESKTOP-MLPLCFD: ~$  
module F = Format  
  
(*TestCase*)  
  
let rec print_list1 fmt il =  
  match il with  
  | [] -> F.fprintf fmt ""  
  | h::t -> F.fprintf fmt "%d;%a" h print_list1 t  
  
let rec print_list2 fmt il =  
  match il with  
  | [] -> F.fprintf fmt ""  
  | (a,b)::t -> F.fprintf fmt("(%d,%d);%a" a b print_list2 t  
  |  
  
let _ =  
  let open TailOpt in  
  let (a, b) = split [(1,6); (2,7); (3,8); (4,9); (5,10)] in  
  let _ = F.printf "(a : [%a], b: [%a])\n" print_list1 a print_list1 b in  
  let c = combine [1;2;3;4;5;] [6;7;8;9;10] in  
  F.printf "c : [%a]\n" print_list2 c  
~
```

[main.ml 내부에 구현된 코드]

```
k0s0a7@DESKTOP-MLPLCFD: ~/week3/hw2$ dune exec ./main.exe  
(a : [1;2;3;4;5;], b: [6;7;8;9;10;])  
c : [(1,6);(2,7);(3,8);(4,9);(5,10);]  
k0s0a7@DESKTOP-MLPLCFD: ~/week3/hw2$
```

[main.exe 실행 결과]

● main.ml 파일은 이번 과제를 구현하고 결과를 출력하기 위해 이전 과제들의 출력파일들을 참고하여 구현하였다. 일반 리스트를 출력하는 print_list1, 튜플형태의 원소를 가지는 리스트를 출력하는 print_list2를 선언하고 그 아래에서 TailOpt의 split 과 combine함수를 호출하여 리스트들을 출력해주었다.

● split함수에 인자로 [(1,6);(2,7);(3,8);(4,9);(5,10);] 인 튜플형태의 원소를 가지는 리스트를 인자로 주었다. 그 후 출력을 했을 때 원하는 결과인 (1;2;3;4;5;) (6;7;8;9;10;)을 얻을 수 있었다.

● combine함수에서도 (1;2;3;4;5;) (6;7;8;9;10;) 두 리스트를 인자로 넘겨주니 [(1,6);(2,7);(3,8);(4,9);(5,10);] 라는 원하는 합쳐진 결과를 얻을 수 있었다.

<과제 구현 중 어려웠던 부분 및 해결 과정>

```
let rec split_tail il acc1 acc2 =  
  match il with  
  | [] -> (acc1, acc2)  
  | (a,b) :: t -> (split_tail t (acc1 @ [a;]) (acc2 @ [b;]))  
  
let rec combine_tail il1 il2 acc =  
  match il1,il2 with  
  | [],[] -> acc  
  | h1::t1, h2::t2 -> combine_tail t1 t2 (acc @ [(h1,h2);])  
  | ([],_::_) | (_::_,[]) -> failwith "Unsupported"
```

[수정하기 전 tailOpt.ml 내부에 구현된 코드]

● 이번 과제에서 초기 단순히 기존 함수들을 tail_recursion 하게만 수정해주는 것으로 착각을 하였다. 그리하여 위의 사진과 같이 기존 함수에서 타입이 바뀐 split_tail 함수와 combine_tail 함수를 정의하였다. 이렇게 사용하는 경우 함수의 사용법이 바뀌어 main 문에서의 호출 방식이 바뀌게 되어 좋지 못한 코딩이 되게 되었다. 이에 잘못된 점을 바로 잡고자 래퍼 함수를 이용하여 가장 바깥쪽에 기존 함수와 타입이 같도록 선언해주어 위의 코드에서의 문제점을 보완하였다. 이때, 래퍼 함수를 사용하여 타입을 유지할 경우 래퍼 함수 자체에서 내부에 tail_recursive하게 구현된 함수를 호출해줘야 하는데 이때 기존의 타입이 같은 인자를 제외한 인자 (Ex. acc) 에는 값을 정의해줌으로써 타입을 유지하였다. 아래는 위의 코드를 보완한 최종 tailOpt.ml 파일의 내부 코드이다.

```
let split il =  
  let rec split_tail il acc1 acc2 =  
    match il with  
    | [] -> (acc1, acc2)  
    | (a,b) :: t -> (split_tail t (acc1 @ [a;]) (acc2 @ [b;]))  
  in  
  split_tail il [] []  
  
let combine il1 il2 =  
  let rec combine_tail il1 il2 acc =  
    match il1,il2 with  
    | [],[] -> acc  
    | h1::t1, h2::t2 -> combine_tail t1 t2 (acc @ [(h1,h2);])  
    | ([],_::_) | (_::_,[]) -> failwith "Unsupported"  
  in  
  combine_tail il1 il2 []
```

[수정된 tailOpt.ml 내부에 구현된 코드]

```

let combine il1 il2 =
  let rec combine_tail il1 il2 acc =
    match il1,il2 with
    | [],[] -> acc
    | h1::t1, h2::t2 -> combine_tail t1 t2 (acc @ [(h1,h2);])
    | ([],_::_) | (_::_,[]) -> failwith "Unsupported"
  in
    combine_tail il1 il2 []

```

[tailOpt.ml 파일 내부에 구현된 combine 함수의 코드]

● combine함수는 이번 과제에서 가장 애를 먹었던 부분인 것 같다. 사실 pattern을 매칭할 때 하나의 인자만 쓸수 있다고 알았다. 머릿속에서는 il1 il2 두 인자에 대해 패턴매칭을 하면 내가 생각하는대로 코딩이 될텐데 하면서도 하나로만 하려고하니 생각이 잘 되지 않았다. 그래서 결국 ocaml에서 두가지 인자에 대해 패턴매칭하는 경우를 여러개 찾아보다가 위와 같이 , (쉼표)를 이용하여 두가지 경우에 대하여 모든 경우를 매칭해주기만 하면 되었던 것이다. 사실 이 방법을 찾느라 1시간정도를 고민하고 검색했던 것 같다.

● 그렇게 패턴매칭에 대해 정의를 해주고 이제 기존에 선언해 두었던 main.ml을 컴파일 해보니 에러가 나와있었다. 알고보니 모든 패턴에 대해 매칭하지 않았다는 이야기였다. 사실 과제에서 주어지길 combine 함수의 경우 인자로 주어지는 두리스트의 길이가 같다고 가정하여 두 리스트의 길이가 다른 경우에 대해서는 고려를 해주지 않았다는 것을 알았고, 위의 사진에서 보이듯이 ([],_::_) | (_::_,[]) 인 패턴을 추가해줌으로써 모든 경우에 대하여 매칭을 해주니 원하는 결과를 얻을 수 있었다.

● 추가적으로 초기 리스트에 원소를 추가하는 방법에도 고민을 많이 했다. ::을 이용하여 원소를 추가해주면 앞에서부터 원소를 추가해주면서 재귀적으로 반복하니 결과가 역순으로 출력되었다. @ 연산을 사용해야겠다 생각하고 진행하였지만 뭔가 오류가 계속 났다. 이 연산은 두 개의 리스트에 대한 연산이라는 기억이 났다. 그래서 @ 연산 전에 추출한 원소들을 하나의 리스트로 만들어서 acc에 @연산을 이용해 연결해주었더니 문제들이 해결되었다. 정말 기분이 좋았다.

<최종 제출 코드 및 결과 화면>

```
k0s0a7@DESKTOP-MLPLCFD: ~$  
let split il =  
  let rec split_tail il acc1 acc2 =  
    match il with  
    | [] -> (acc1, acc2)  
    | (a,b) :: t -> (split_tail t (acc1 @ [a;]) (acc2 @ [b;]))  
  in  
    split_tail il [] []  
  
let combine il1 il2 =  
  let rec combine_tail il1 il2 acc =  
    match il1, il2 with  
    | [], [] -> acc  
    | h1::t1, h2::t2 -> combine_tail t1 t2 (acc @ [(h1,h2);])  
    | ([], _::_) | (_::_, []) -> failwith "Unsupported"  
  in  
    combine_tail il1 il2 []
```

[tailOpt.ml 내부에 구현된 코드]

```
k0s0a7@DESKTOP-MLPLCFD: ~/week3/hw2$ dune exec ./main.exe  
(a : [1;2;3;4;5;], b: [6;7;8;9;10;])  
c : [(1,6);(2,7);(3,8);(4,9);(5,10);]  
k0s0a7@DESKTOP-MLPLCFD: ~/week3/hw2$
```

[main.exe를 이용하여 tailOpt.ml을 테스트한 결과]

- tailOpt의 내부 코드를 보면 split과 combine 두 함수 모두 초기 제시된 각각의 타입을 래퍼함수를 통해 유지하고 있으며, 각각의 내부 *_tail 함수들은 tail-recursive하게 구현되어 있음을 알 수 있다.
- 또한 이에 대하여 main.ml을 컴파일하고 실행해본 결과 원하는 결과를 얻을 수 있었다.
- 이렇게 tail-recursive하게 구현하였을 때의 장점은 call-stack이 기존의 재귀함수보다 적게 쌓인다는 장점이 있다. 재귀가 반복될 때마다 이전 function-call로 돌아갈 일이 없도록 acc를 이용하여 연산을 진행하면서 구현하면 call-stack에 필요없는 function-call은 쌓이지 않게 되고, 이를 통해 call-stack overflow exception을 피할 수 있다고 배웠다. 이번 실습을 통해 이론에서의 내용을 더욱 잘 이해할 수 있었던 것 같다.

<과제를 하며 어려웠던 부분 및 새로 고찰한 내용>

● 이번 과제를 하며 어려웠던 부분으로는 여러 가지가 있었다.

1. 초기 tail-recursive하게 구현하면서 함수의 타입을 지켜주지 못했음.

→ 래퍼함수를 이용해 함수의 타입을 유지시켜줄 수 있었음.

2. 리스트의 뒤에 차곡차곡 쌓아주려면 :: 연산 외에 다른 방법을 찾아야 했음.

→ @ 연산을 이용하되, 해당 연산은 두 개의 리스트 타입간의 연산으로 담고 싶은 원소를 한 개짜리 리스트로 생성하여 @ 연산을 사용하여 해결할 수 있었음.

3. pattern-matching을 할 때 한 개의 패턴만 matching 가능한줄 알고있어 애를먹음.

→ 찾아보니 두 개의 패턴을 비교할 수 있는 방법이 있었고, ‘,’(쉼표)를 이용하여 사용함.

4. 두 개의 패턴에 대해서 비교가능한 패턴을 조금 더 고려하지 못하고 컴파일시 에러로 알게 되었음.

→ matching할 요소가 한 개에서 두 개로 늘어났으면 그에 대해서도 pattern이 생김을 고려하지 못하고 컴파일시에 깨달았음. 해당 pattern의 경우를 추가해주었고, 이후 컴파일시에 오류가 나지 않았음.

위와 같은 어려움이 있었지만 이런 오류들을 해결할수록 내 것으로 만드는데 정말 많은 도움이 되었고, 점점 발전하는 스스로를 보면서 더욱 열심히 하게 되는 것 같다.