Project
SNU 4910.210, Fall 2015
Chung-Kil Hur
due: 12/20 (Sun) 23:59

Problem 1 (30 %)

Racket의 expression을 입력 받아 결과값을 계산하는 인터프리터 함수 `myeval`을 Racket으로 직접 구현하라.

$$\texttt{myeval} : E \to V$$

- 입력은 아래에 주어지는 규칙에 따라 조합된 E의 앞에 quote(')가 붙어 들어온다. ex) `'(cons 3 5)`
- 출력은 숫자, boolean, pair의 경우 Racket에서 제공하는 값을 사용하며, lambda 값의 경우는 각자가 임의로 정의한다.
- 입력이 적절하게 처리될 수 없는 경우 예외처리를 통해 실행을 중단하고 알맞은 메시지를 출력해야 한다.
- Racket에서 제공하는 함수 `eval`은 사용할 수 없다.

C	::=	n	정수
	|	`#t`	참
	|	`#f`	거짓
	|	`'()`	NULL

E	::=	C	상수
	|	x	변수
	|	`(if` E E E`)`	conditional
	|	`(cons` E E`)`	pair
	|	`(car` E`)`	car
	|	`(cdr` E`)`	cdr
	|	`(lambda (`x^*`)` E`)`	function
	|	`(`E E^*`)`	application
	|	`(let ((`x E`)`*`)` E`)`	let
	|	`(letrec ((`x E`)`*`)` E`)`	letrec
	|	`(+` E E`)`	addition
	|	`(-` E E`)`	subtraction
	|	`(*` E E`)`	multiplication
	|	`(=` E E`)`	equality
	|	`(<` E E`)`	less than
	|	`(>` E E`)`	greater than

실행 예제

- `(myeval '(let ((p (cons 1 (cons 2 '()))))) (cons 0 p)))`
  결과: `'(0 1 2)`
- `(myeval '(letrec ((f (lambda (x)`
  `(if (= x 0) 0 (+ x (f (- x 1)))))))) (f 5))`
  결과: `15`
- `(myeval '((lambda (f) (lambda (x) (f x))) (lambda (x) (+ x 1))))`
  결과: `'((lmda (x) (f x)) env ...)` (각자 임의로 정의 가능)

Problem 2 (5 %)

Racket expression을 입력 받아 OCaml에 정의된 Racket 문법으로 변환하는 파서를 재귀함수로 직접 구현하라. Racket 문법의 정의가 포함된 파일 (`syntax.ml`)과, Racket expression 문자열을 토큰으로 구분해주는 렉서는 조교가 제공한 것을 사용한다. 파서 구현에 실패한 경우, 이후 문제에서는 조교가 OCaml 바이트코드 바이너리 파일로 제공하는 파서를 사용할 수 있다.

Problem 3 (10 %)

앞에서 구현한 파서를 사용하여, 1번 문제에서 구현한 `myeval`을 OCaml로 구현하라.

Problem 4 (5 %)

`myeval`을 끝재귀(tail-recursion) 최적화를 이용해 효율적으로 구현하라. 아래 첫 번째 예시와 같이 최적화를 사용할 수 있는 입력은 `myeval`이 올바르게 처리할 수 있어야 한다. 두 번째 예시는 최적화를 적용할 수 없는 예제이다.
힌트: OCaml의 tail call 최적화 덕을 보도록 여러분이 구현하는 `myeval`을 잘 구현하면 된다.

실행 예제

- `myeval "(letrec ((f (lambda (x n)`
  `(if (= x 0) n (f (- x 1) (+ n x)))) )) (f 999999 0) )"`
  결과: `499999500000`
- `myeval "(letrec ((f (lambda (x)`
  `(if (= x 0) 0 (+ x (f (- x 1))))) )) (f 999999))"`
  결과: 오류(`Stack overflow during evaluation (looping recursion?).`)

Problem 5 (10 %)

앞에서 구현한 `myeval`에 mutable pair를 추가하라. mutable pair를 생성하는 `mcons`, 값을 꺼내오는 `mcar`과 `mcdr`, 저장된 값을 변경하는 `set-mcar!`, `set-mcdr!`를 구현하여야 한다.

E ::= ...

|	`(mcons` E E`)`	mutable pair
|	`(mcar` E`)`	mutable car
|	`(mcdr` E`)`	mutable cdr
|	`(set-mcar!` E E`)`	change first elem
|	`(set-mcdr!` E E`)`	change second elem

실행 예제

- `myeval "(let ((mp (mcons 1 2))) (mcar mp))"`
  결과: `1`
- `myeval "(let ((mp (mcons 1 2)))`
  `(let ((tmp (set-mcdr! mp 3))) (mcdr mp)))"`
  결과: `3`

Problem 6 (10 %)

`myeval`에 예외 처리 기능을 추가하라. `raise`를 이용해 예외를 발생시키고, `with-handlers`가 예외를 처리한다.

E ::= ...

|	`(raise` E`)`	raise exception
|	`(with-handlers ((`E E`)`$^+$`)` E`)`	exception handler

- `myeval "(with-handlers`
  `(((lambda (x) (= x 5)) (lambda (x) (* x 2))))`
  `(cons (+ 1 3) (- 2 (raise 5))))"`
  결과: `10`

Problem 7 (30 %)

`lambda` 함수의 실행을 최대한 최적화하는 새로운 인터프리터 함수 `myeval_memo`를 구현하라. 각자 스스로의 방법을 통해 코드를 분석하여 기억화(memoization) 기법을 적용하되, `myeval_memo`의 실행 결과는 언제나 `myeval`과 같아야 한다. 기억화가 적용된 함수는 실행될 때마다 함수 인자와 결과값을 해시테이블(Hash Table)에 저장하며, 이 후 같은 인자로 다시 불릴 경우 그 함수를 실행하지 않고 해시 테이블에 미리 기억에 놓은 결과값을 가져와서 바로 반환한다. 해시 테이블은 OCaml에서 제공하는 `Hashtbl`을 사용할 수 있다.

기억화는 값-중심 함수(즉, 메모리를 변경시키지 않거나 메모리 변화에 영향을 받지 않는 함수)에만 적용해야 한다. 그렇지 않다면 기억화된 함수가 잘못된 결과를 내어 놓을 수 있다.

최대한 많은 함수에 기억화가 적용될수록 좋은 점수를 받을 것이다. 단, 기억화를 값-중심이 아닌 함수에 적용했을 경우에는 잘못된 결과를 낼 수 있어 오히려 감점될 수 있음을 유의하라.

실행 예제

- `myeval_memo "(letrec ((fib`
  `(lambda (n) (if (= n 0) 0 (if (= n 1) 1`
  `(+ (fib (- n 1)) (fib (- n 2))) ))) )) (fib 100))"`
  결과: `1298777728820984005`

- `myeval_memo "(let ((x 1))`
  `(let ((f (lambda () x)))`
  `(let ((y (f))) (let ((tmp (set! x 5))) (+ y (f))))))"`
  결과: 6