2017년 1학기
프로그래밍언어개론

- Cute17 project Item2 -

팀 명 : 레온고?
조 장 : 201203405 류치현
팀 원 : 201202149 백승진

1-1 구현 일정 계획

작업내용	6/1	6/2	6/3	6/4	6/5	6/6	6/7
목표 설정							
요구사항 이해							
역할 분담 및 구현							
주석 및 리팩터링							
보고서							

1. 목표 설정
팀 결성과 동시에 전반적인 작업 방향과 목표 설정
- git hub를 이용한 협업 능력 배양
- Racket을 이용한 테스트 케이스의 의미 파악

2. 요구사항 이해
테스트 케이스의 의미 파악과 더불어 구현해야하는 함수 구체화
- 크게 구현해야할 두 가지는 lambda와 define
- 테스트 케이스 1-7/ 8-16/ 17/ 18-19/ 20 으로 나누어 세분화

3. 역할 분담 및 구현
크게 나뉜 함수를 기반으로 역할 분담 및 구현
- define과 lambda에 대한 내용을 각자의 생각에 따라 구현
- 구현과 동시에 함수의 내부 구현 및 디테일한 부분을 공유

4. 주석 및 리팩터링
보고서 작성을 위한 상기 작업과 주석, 리팩터링 등 마무리 작업
- 기본적으로 구현 과정에서 주석을 달았지만, 정리 및 간략화
- 코드에서 어수선한 코드, 불필요한 부분 제거

5. 보고서 작성
지금 까지 구현한 내용을 바탕으로 양식에 맞는 보고서 작성

1-2 팀 구성

학번	이름	역할		
		함수구현	주석 및 리팩터링	보고서
201203405	류치현	define/17/20	리팩터링	공동 작업
201202149	백승진	lambda/18/19	주석	공동 작업

* 팀장 (조교가 연락을 취할 수 있는 대표) : 류치현(010-8633-8952)

1-3 수정 로그
git 주소: https://github.com/ChiHyeonRyu/PL-Item2.git (자세한 로그는 git 참조 요망)
6월 5일 - 테스트 19번까지 완료 및 코드 수정
6월 6일 - 모든 테스트 완료
- 코드 리팩토링: 코드 가독성을 위함.
- 주석 첨부 및 최종 코드 정리

2-1 채택/고안한 내용, 기법에 대한 설명

> Define 구현

Define 구현을 위해 Define() 함수를 run_func() 함수에 정의하였다. 우선 define 문의 형식에 따라 define 할 변수와 값을 각각 노드로 분리한다. 분리한 변수와 값 노드를 전역 심볼테이블에 저장하기 전에, **값이 가질 수 있는 타입에 따라 조건문을 구성하여 테이블에 넣을 수 있도록 한다.** 값이 변수일 경우에는 그 변수에 해당하는 값을 심볼테이블에서 찾아와 대체하도록 한다. 값이 수식일 경우 해당 수식을 계산 한 후 최종 값으로 대체한다. 최종적으로 define할 변수와 값을 테이블에 저장한다.

> Lambda 구현

Lambda 구현을 위해 run_lambda() 함수를 run_func() 함수에 정의하였다. 우선 람다 함수의 처리는 두 종류로 나눌 수 있다. 람다 함수 자체가 **①실인자화 함께 입력되는 경우**와 람다 함수를 **②변수에 바인딩 한 후 변수를 통해 실인자를 주어 처리하는 경우**가 있다. ①의 경우 수식이 ((lambd ~)의 형태이기 때문에, run_list() 함수를 재귀 호출함으로서 람다식의 이중으로 된 리스트를 벗긴다. 이후 run_func() 함수를 실행 할 수 있도록 한다.

run_lambda :

본격적으로 run_lambda를 보면, lambda라는 키워드를 가진 리스트들의 가독성을 좋게 하기 위해 각 변수에 맵핑시켜 주었다.

formal_param_node	함수에서 인자를 받아 임시 저장하는 변수 (매개변수)
exp_node	함수 안의 내용 (람다 함수의 실행 부분)
actual_param_node	함수에서 사용되는 실인자의 값

이 변수들을 전역으로 이용하기 위해 symbolTable을 global로 선언하고 그 테이블 안에 key: formal_param_node, value: actual_param_node로 저장하여 사용한다.

실인자 없이 람다 함수 자체만 입력 될 경우 해당 람다 함수 자체를 그대로 리턴 하도록 한다. 매개변수를 while 반복문을 통해 개수에 상관없이 실인자로 바인딩 한 후 심볼테이블에 저장해준다. 이후 실행부분을 run_expr() 함수를 통해 실행 한 후 결과를 반환한다. 과제를 진행하면서, 20번 테스트 코드를 보고 실행 부분이 여러개 될 수 있다는 사실을 알았고, 실행 부분의 끝까지 수행을 할 수 있도록 하였다. 함수 종료 후의 함수의 매개변수(지역변수) 삭제는 심볼테이블의 백업을 통해 구현하였다. (자세한 설명은 변수의 scope 부분 설명을 참조)

> 변수의 바인딩 처리

```
#심볼테이블에 넣기
def insertTable(id, value):   id: 'a'   value: [INT:1]
    symbolTable[id] = value
    return symbolTable[id]
```

: 전역으로 선언된 심볼테이블에 변수를 바인딩 하는 모습이다. define() 함수에서 분리된 노드를 이용하여 테이블에 삽입하며, 변수는 string 타입으로, 값은 노드 타입으로 저장하여 바인딩 하였다.

```
#심볼테이블에서 값 찾기
def lookupTable(id):   id: 'a'
    if id in symbolTable:   #심볼테이블에 있으면 해당 값 리턴
        v_node = symbolTable[id]   v_node: [INT:1]
        return v_node
    else:   #없으면 None 리턴
        return None
```

↓ 변수에 바인딩 된 값 찾기 (run_expr())

```
#ID를 이용해서 테이블에서 값(노드) 찾기
if root_node.type is TokenType.ID:
    root_node = lookupTable(root_node.value)
    return root_node
```

변수의 값을 찾는 경우는 우선 run_expr() 함수에서, 토큰타입이 'ID'일 경우 해당 변수를 lookupTable() 함수 호출을 통하여 define한 변수의 값을 찾는다. 변수에 해당 하는 값 노드를 반환하며 이후 run_func()의 각 연산자 함수들로부터 run_expr() 함수의 호출을 통해 저장된 변수의 값을 구하는 것이 가능하다.

> 함수의 바인딩처리

```
#심볼테이블에 넣기
def insertTable(id, value):   id: 'plus1'   value: ([LAMBDA] ([ID:x]) ([PLUS] [ID:x] [INT:1]))
    symbolTable[id] = value
    return symbolTable[id]
```

```
#심볼테이블에서 값 찾기
def lookupTable(id):   id: 'plus1'
    if id in symbolTable:   #심볼테이블에 있으면 해당 값 리턴
        v_node = symbolTable[id]   v_node: ([LAMBDA] ([ID:x]) ([PLUS] [ID:x] [INT:1]))
        return v_node
    else:   #없으면 None 리턴
        return None
```

```
insertTable(formal_param_node.value, actual_value_forBinding)   #매개변수에 바인딩
```

: 변수의 바인딩과 마찬가지로 define() 함수를 통해 변수에 람다 함수를 바인딩하여 심볼테이블에 삽입하여 준다. 이후 전역 함수 호출(plus1 3)을 해줄 경우, 심볼테이블에서 해당 변수(plus1)에 저장된 람다 함수를 반환, 실행하게 된다. run_lambda() 함수에서 분리된 람다식의 실행부분(+ x 1)을 insertTalbe() 함수 호출로 실인자(3)을 매개 변수(x)에 바인딩한다. 이후 run_expr() 함수 호출로 람다식 실행부분을 수행하게 되고 결과 값을 얻는다.

※ 변수 scope 구현 - 지역변수의 처리 방법 (SSR 방식의 구현)

```
global symbolTable  #전역 심볼테이블을 사용하기위한 선언

backupTable = symbolTable.copy()     #현재 scope 참조환경 백업

if actual_param_node is None:    #첫번째 원소가 람다일때 두번째 원소가 없으면 그대로 리턴
    return node
```

```
#실행부분 실행. 실행부분이 한개 인 경우
result = run_expr(exp_node)

#실행부분이 두개 이상인 경우 실행부분을 끝까지 수행
while exp_node.next is not None:
    result = run_expr(exp_node.next)
    exp_node = exp_node.next

symbolTable = backupTable     #이전 참조환경으로 복원

return result    #람다식의 결과를 리턴
```

: 지역 변수의 처리방법에 대해 많은 고민을 하였다. 부모 프로그램의 변수의 이름과 자식 프로그램의 변수 이름이 같을 경우, 자식 프로그램이 종료 되었을 때 변수의 삭제, 부모 프로그램 변수의 복원이 필요했다. 초기에는 각 매개변수를 임시로 담을 리스트를 구성하고 미리 부모 프로그램의 변수를 백업한 후, 함수의 종료 직전에 복원해 주는 방법을 구상하였다. 하지만 그러한 방식을 사용할 경우, 20번 테스트 케이스 Nested 함수에서 define 된 내부 프로그램이 함수 종료 후 심볼 테이블에서 사라지지 않는 문제가 발생했다.

이후 고민을 거듭한 끝에 심볼테이블 자체를 백업하고 복원하는 방식을 이용해 보았다. 람다 함수가 실행 될 때, 이전 scope의 참조 환경을 백업한다. 즉 심볼테이블 하나가 추가되어 현재 실행되는 함수만을 위한 참조환경을 가질 심볼테이블이 생성 되었다는 의미가 된다. 이는 심볼테이블에 임시로 쓸 지역변수를 바인딩하는 것과 같다. 이 후 실행되는 코드는 자신의 참조 환경에서 동작하며, 함수의 종료 직전 자신의 참조 환경을 이전 참조 환경으로 복원함으로서 지역변수의 삭제 기능을 하게 된다.

자식 서브프로그램의 참조환경(심볼테이블) 변경은 부모 참조환경에는 영향을 못 미친다.

variable	value
a	10
b	20

* 부모 서브프로그램 참조환경

함수 시작

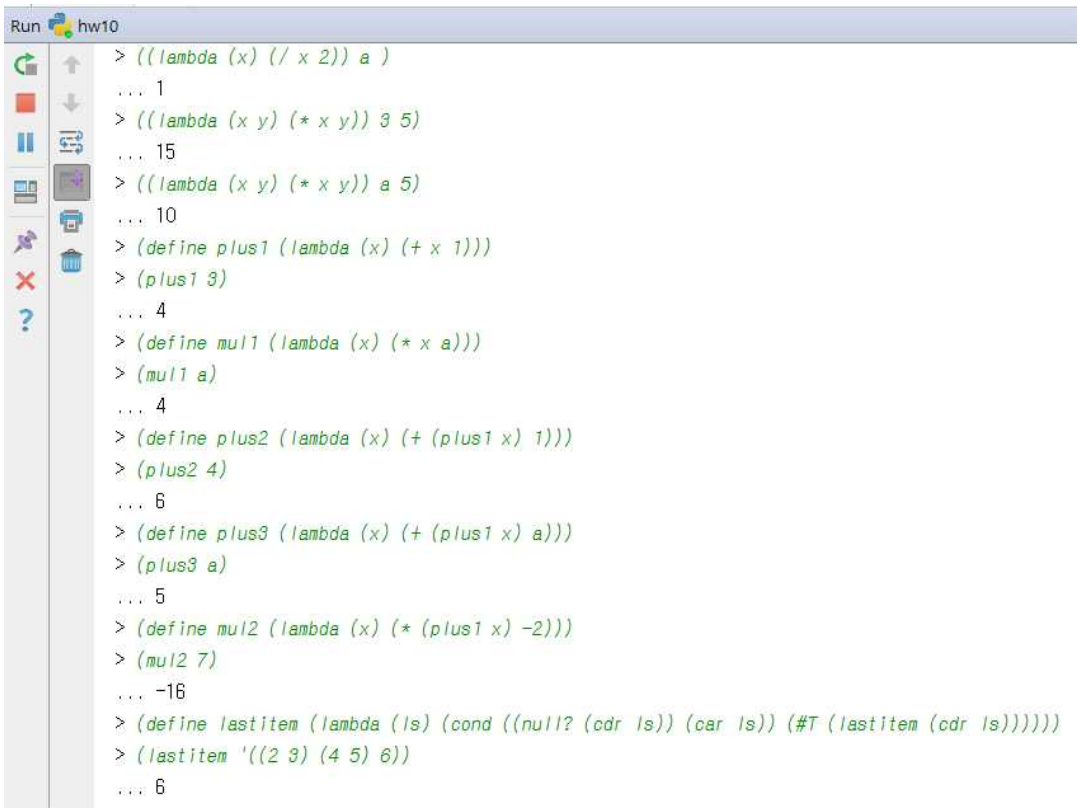
함수 종료

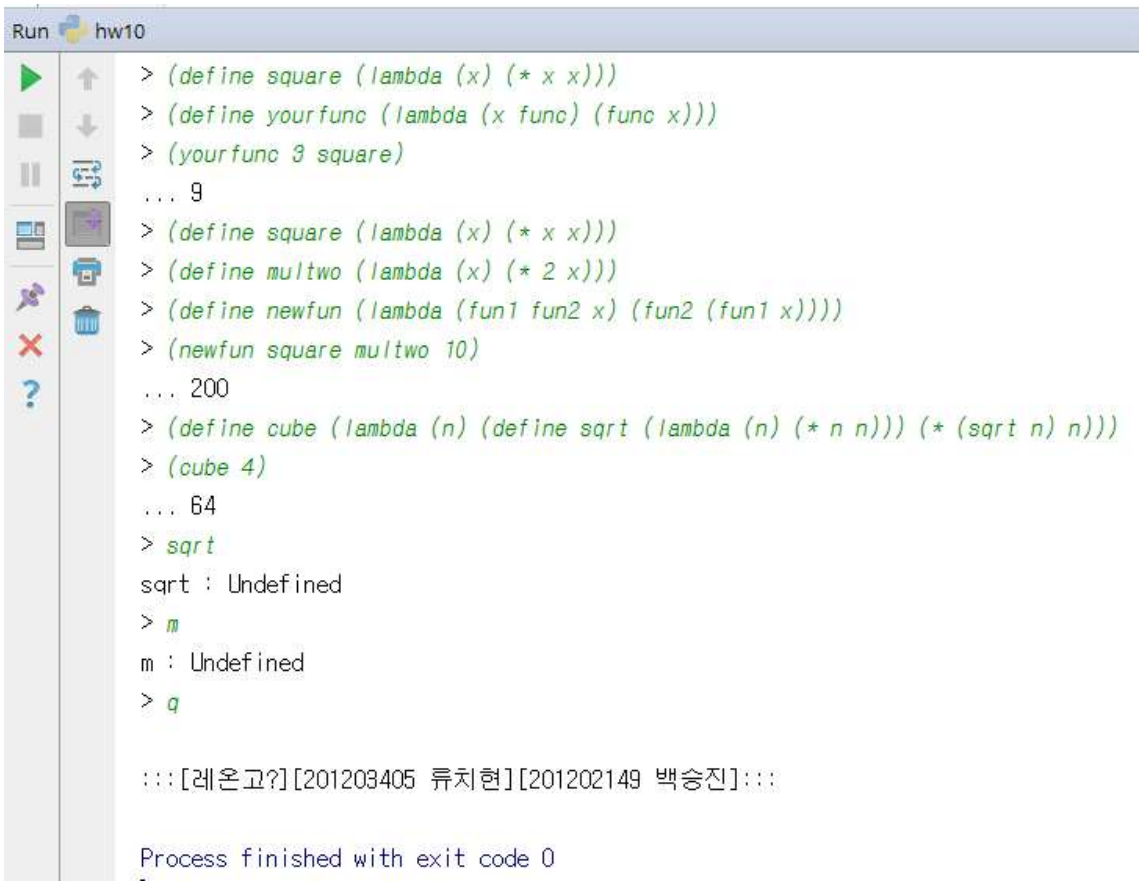
variable	value
a	5
b	20
c	3

* 자식 서브프로그램 참조환경

2-2 결과 요약


Run
hw10
C:₩Python27₩python.exe C:/Users/류치현/git_workspace/PL-Item2/PL-Item2/hw10.py
::: 종료를 원하면 'q' 혹은 'Q'을 입력하세요 :::
> (define a 1)
> a
... 1
> (define b '(1 2 3))
> b
... '(1 2 3)
> (define c (- 5 2))
> c
... 3
> (define d '(+ 2 3))
> d
... '(+ 2 3)
> (define test b)
> test
... '(1 2 3)
> (+ a 3)
... 4
> (define a 2)
> (* a 4)
... 8
> ((lambda (x) (* x -2)) 3)
... -6


Run
hw10
> ((lambda (x) (/ x 2)) a)
... 1
> ((lambda (x y) (* x y)) 3 5)
... 15
> ((lambda (x y) (* x y)) a 5)
... 10
> (define plus1 (lambda (x) (+ x 1)))
> (plus1 3)
... 4
> (define mul1 (lambda (x) (* x a)))
> (mul1 a)
... 4
> (define plus2 (lambda (x) (+ (plus1 x) 1)))
> (plus2 4)
... 6
> (define plus3 (lambda (x) (+ (plus1 x) a)))
> (plus3 a)
... 5
> (define mul2 (lambda (x) (* (plus1 x) -2)))
> (mul2 7)
... -16
> (define lastitem (lambda (ls) (cond ((null? (cdr ls)) (car ls)) (#T (lastitem (cdr ls))))))
> (lastitem '((2 3) (4 5) 6))
... 6



: 20번 테스트 케이스까지 올바른 결과를 보인다. 지역변수 x, n, ls 등을 입력시 Undefined 문장을 출력한다. sqrt 또한 삭제 된 변수(Undefined) 이므로 (sqrt 4)는 수행되지 않는다.

: 20번 테스트 까지 완료 한 후 심볼테이블을 보면, n, x, ls, sqrt 등의 임시로 할당 된 변수(지역변수)들이 사용된 후 정상적으로 소멸되어 전역적으로 define한 변수들만 남은 것을 볼 수 있다.

2-3 장단점 분석

최초 지역변수 처리를 위한 구현을 할 때에는 심볼테이블 자체의 저장과 복원 방법을 생각하지 못하였는데, 많은 오류를 내었다. 이 후 20번 테스트 구현을 위해 참조 환경을 저장, 복원하는 방식을 구현하고 보니 많은 코드를 줄일 수 있었고, 보다 직관적인 코드가 되었다.
최대한 기존 코드의 변경을 최소화하며 구현하였기 때문에 어느 테스트 환경에서든 올바르게 동작할 수 있도록 구성되었다고 보인다.

3. 느낀점

류 치 현 :
디버깅을 굉장히 많이 한 과제였다. 코드양이 많은 편이고, 함수의 호출 관계가 다소 복잡해서 후반부 테스트 케이스를 구현하는 것에는 많은 시간을 할애했다. 지역 변수의 처리 방식에 대해 많은 고민을 하였고, 구현하는데에 이론 수업에서 배운 지식이 많은 도움이 되었다. 과제 수행에 있어서 git의 사용 또한 협업 및 코드 백업에 있어서 굉장히 효율적이었다. 힘들었던 과제였지만 완성하고 나니 굉장한 뿌듯함을 느꼈고, 학기 시작할 때에만 해도 사용법조차 몰랐던 파이썬 코딩 또한 이제는 어느정도 능숙해진 것 같다.

백 승 진 :
실전 코딩에서 배워온 git를 실질적으로 사용해보는 경험을 할 수 있었고, 과제를 진행하면서 막히는 부분이 과거에 진행해온 작업의 일부가 많음을 확인 할 수 있는 기회가 되었습니다.
그에 따라 베이스 작업이 얼마나 중요하고 범용적으로 사용해도 문제가 되지않는 코드가 얼마나 좋은 것인지를 몸으로 체감할 수 있는 과제였습니다.
또, 과제를 진행하면서 예상했던 대로 인터프리터를 결국 구현하게 되었는데, 차근차근 과제를 쫓아오다보니 어느새 완성하고 있는 모습을 보면서 자신감과 보람을 느낄 수 있었습니다.