2017년 1학기
프로그래밍언어개론 설계 계획서

(팀이름) : 개인
(조장) : [201504280] [신윤호]
<작업의 분할>
개인 과제 수행

1-1 구현 일정 계획

작업내용	29	30	31	1	2	3	4	5	6
바인딩 구현	■								
변수의 사용 및 scope구현		■							
lambda구현			■	■					
전역 함수 호출 구현					■				
함수 내에서 전역함수 호출 등 다양한 전역 함수 호출 방법 구현					■				
recursion구현						■	■		
nasted구현								■	■
함수에서 함수 인자 사용 구현						■			

1-2 팀 구성

학번	**이름**	**역할**
201504280	신윤호	개인 과제 수행

* 팀장 (조교가 연락을 취할 수 있는 대표) : 신윤호

2017년 1학기
프로그래밍언어개론 설계 (중간, 최종) 보고서

조번호(팀이름) : 201504280 (신윤호)

2-1 채택/고안한 내용, 기법에 대한 설명

1. 변수 및 함수의 바인딩(define 구현)

변수 및 함수의 정의는 var_def함수를 이용해 수행하였다. define키워드를 만나면 var_def가 변수 또는 함수를 전역 테이블인 varTable에 저장한다. 이 때, 변수의 값으로 올 수 있는 것은 상수와 Quote리스트, 그리고 함수가 올 수 있다. 변수는 딕셔너리 자료형으로 구현한 전역 테이블인 varTable에 전역으로 저장된다.

함수도 기본적으로 변수와 마찬가지로 varTable에 저장된다. 단, 함수의 경우에는 변수값으로 '함수 테이블'이 저장된다. 함수 테이블은 2개 이상의 필드를 가지고 있다. expr은 함수의 수식을 저장하는 필드이고, 나머지 필드는 매개변수를 저장하는 필드이다. 매개변수의 개수에

따라 매개변수 필드는 늘어날 수 있다. 그리고 nasted함수의 경우 nasted여부를 표시하는 nasted필드를 가지게 된다. 함수 테이블은 일반 dictionary자료형이 아닌, collections의 ordered dictionary 자료형을 사용했다. 필드들을 일정한 순서대로 저장하기 위함이다.

varTable의 전체적인 모습은 다음과 같다.

varTable 구조

<table>
<tr><th>변수명</th><th>변수값</th></tr>
<tr><td>x</td><td>10</td></tr>
<tr><td>func</td><td>func에 대한 함수테이블
<table>
<tr><td>expr</td><td>(lambda (x) (+ x y))</td></tr>
<tr><td>x</td><td>None</td></tr>
<tr><td>y</td><td>None</td></tr>
<tr><td>nasted</td><td></td></tr>
</table></td></tr>
<tr><td>...</td><td>...</td></tr>
</table>

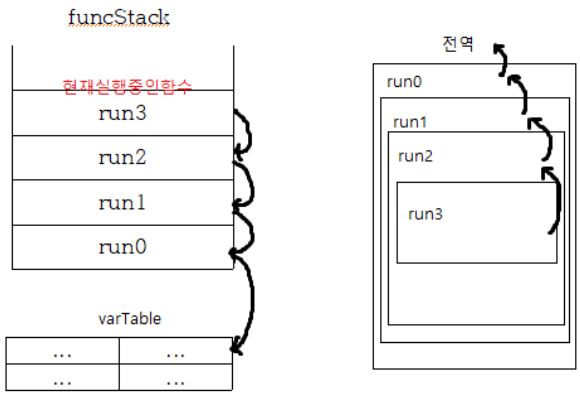
2. 변수 값의 사용

변수 값의 사용은 var_get함수를 이용해 구현하였다. var_get함수는 정적 스코프 룰(SSR)에 따라 변수를 찾게 된다. 스코프 구현에 관한 사항은 관련된 항목에서 더 자세히 기술하겠다. var_get함수는 테이블에서 변수명을 key값으로 하는 value를 찾아서 반환한다. 이때, 만약 변수가 함수를 저장하고 있는 경우 해당 함수의 함수 테이블을 반환한다.

3. 변수의 Scope구현

변수를 찾는 역할을 하는 var_get함수는 정적 스코프 룰(SSR)에 따라 변수를 찾는다. 맨 처음에는 자신의 영역에서 변수를 찾는다. 만약 자신의 영역에 찾고자하는 변수가 없으면 자신의 static parent의 영역에서 변수를 찾는다. 이러한 방식으로 static parent를 조사하다 마지막으로 전역 변수 테이블인 varTable에서 변수를 찾는다.

더욱 구체적으로 설명하면, 우선 함수가 실행되면 함수는 전역 리스트인 funcStack에 자신의 함수명을 넣는다. funcStack은 스택의 역할을 수행하며 함수가 호출될 때 마다 차곡차곡 쌓이게 된다. var_get함수는 만약 funcStack에 함수가 존재하는 경우, funcStack의 맨 위에 있는 함수의 함수 테이블부터 조사하게 된다. 만약 변수를 찾을 수 없는 경우, 그 다음에 있는 함수 테이블을 조사하게 된다. 즉, 처음에는 자신의 영역에서 변수를 찾고, 그 다음에는 static parent의 영역을 조사하게 되는 것이다. 이런 방식으로 자신의 static parent에 해당하는 모든 영역을 조사한 뒤, 마지막으로 전역 테이블을 조사하게 되는 방식이다. 이 과정을 그림으로 나타내면 다음과 같다.



4. 전역 함수 호출

전역 함수가 호출되면 run_func에서 run_user_func를 호출한다. run_user_func의 기능은 다음과 같다.

① 에러 검사

- 정의되지 않은 전역 함수를 호출할 경우 에러 메시지를 출력한다.

② actual parameter의 처리

- 함수 또는 특정한 연산 결과를 actual parameter로 사용할 수 있도록 하기 위해서 처리하는 부분이다. actual parameter가 INT, ID, QUOTE가 아니라면 run_expr을 수행해 준다.

③ lambda식 생성

- varTable에서 전역 함수에 대한 정보(함수 테이블)을 가져온다. 그리고 전달된 actual parameter를 함수식에 붙인 뒤 리스트로 묶어 lambda식 형태로 만든다. 단, 여기서 파라미터를 바인딩하는 것은 아니다. 바인딩은 lam함수 내부에서 이루어지고, 여기서는 lam함수가 항상 ((lambda (x) (+x 1) 2) 형태의 식을 받도록 하기 위해 (lambda (x) (+x 1)) 형태의 식을 ((lambda (x) (+x 1) 2)형태로 만들어주는 것이다.

④ nasted함수인지 검사

- nasted함수인지 검사하여 적절한 절차를 처리한다. 자세한 부분은 nasted항목에서 설명하도록 하겠다.

⑤ lam함수(lambda처리 함수)를 호출해 결과값을 반환한다. 즉, 전역 함수를 호출하면 run_func가 호출되고, run_func가 run_user_func를 호출해 lambda를 처리할 준비를 마친다. 그리고 lam을 호출해 lambda를 처리하는 흐름으로 전역 함수 호출이 수행된다.

5. lambda구현

lambda는 lambda라는 키워드가 이미 파이썬에 있어서, 이름을 lam으로 하여 정의했다. lam 함수는 다음과 같이 동작한다.

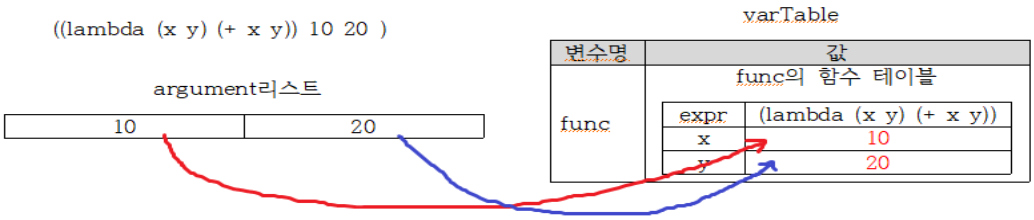
① 이름 없는 함수에 대한 작업

- 이름 없는 함수란 define을 통해 특정한 이름으로 바인딩되지 않은, 사용자가 직접 콘솔에 입력한 lambda식을 말한다. (ex. ((lambda (x) (+ x 1)) 2))
- 이름 없는 함수와 이름을 가진 함수를 구분하지 않고 처리하기 위해, 이름 없는 함수를 null이라는 이름을 지닌 함수로 만든다. 즉, varTable에 이름이 null인 함수의 함수 테이블을 만들어 저장한다.

- null이라는 이름이 중복되면 바인딩 해놓았던 actual parameter가 덮어씌워지는 등의 문제가 발생한다. 따라서 define문에 함수 이름을 null로 정의하지 못하도록 하는 코드를 추가했다. 또한, 함수의 인자로 함수를 사용하는 경우 등 여러 개의 null함수를 생성해야 하는 경우가 있다. 이런 경우에 null함수들이 함수테이블을 공유하게 되면 문제가 발생할 수 있다. 이를 방지하기 위해 null1, null2, null3... 이런 식으로 null뒤에 번호를 붙여 각각의 null함수들을 구분해 주었다.

② formal parameter에 actual parameter바인딩

- actual parameter를 argument라는 이름의 리스트로 받아서, 함수 테이블에 있는 formal parameter의 값을 할당해준다.



- 이 때, 함수에서 함수를 인자로 사용할 수 있기 때문에 actual parameter로 함수 테이블이 전달될 수 있다. 이때는 함수 테이블을 formal parameter에 바인딩 해준다.

③ 함수 몸체(실제 처리될 연산) 노드들을 분리해서 따로 노드로 만듬

④ 함수명을 funcStack에 push

⑤ 함수 실행

- 함수 몸체를 run_expr한다.
- recursion을 구현하기 위해 run_expr후 일련의 조건을 검토한다. 만약 결과에 varTable에 기록된 것이 있고, 그것이 함수라면 결과를 바로 반환하지 않고 run_user_func를 호출해 함수를 재귀적으로 호출한다. recursion 구현에 대한 더 자세한 이야기는 recursion항목에서 하도록 하겠다.

⑥ 함수명 funcStack에서 pop

⑦ 바인딩된 파라미터 초기화

- formal parameter를 None으로 초기화 해준다.
- 익명함수인 null함수인 경우 함수 자체를 삭제한다.

⑧ 결과 값 반환

5. recursion구현

실제 recursive call의 모습을 모사해 recursion을 구현했다. 함수가 실행되면 funcStack에 함수 이름이 추가된다. 만약 함수 실행 도중 다른 함수를 만나면 함수 실행을 잠시 중단하고 해당 함수를 진행한다. 이런 방식으로 funcStack에 함수들의 목록이 쌓이게 된다. 재귀호출의 끝부분에서는 맨 마지막에 스택에 들어온 함수부터 차례대로 return 한다. 그리고 마지막으로는 맨 처음 호출된 함수가 결과를 return하며 종료된다.

funcStack에 함수 테이블을 넣지 않고 단순히 함수 이름만 넣은 이유는 함수 이름을 통해 varTable에서 함수 테이블에 접근할 수 있기 때문이다. 함수 이름만 알면 함수 테이블에 접근 가능하고, 함수 테이블에 접근하면 함수의 몸체와 매개변수에 모두 접근할 수 있기 때문에 함수 이름만 넣게 되었다.

함수 내부에서 또다시 함수 호출을 해야 하는지, 아니면 return할 것인지는 함수 몸체를 run_expr한 결과를 저장한 변수인 result를 조사해서 판단했다. result에 함수가 있는 경우 run_user_func를 호출해 해당 함수를 수행하도록 했다. 이 과정에서 원래 진행 중이던 함수는 자연스레 잠시 중단되고 해당 함수가 수행된다. 만약 해당 함수 내부에도 또 다른 함수가 있는 경우 마찬가지 과정이 반복된다. 그리고 마지막 recursive call에서 결과가 나오게 되고 funcStack에서 함수를 pop하며 결과를 return한다. 이 과정을 맨 처음 호출된 함수가 결과를 return할 때까지 반복한다.

6. Nasted 함수 구현

중첩함수의 경우 식을 define할 때 var_def에서 함수 테이블에 nasted필드를 추가해 주도록 하여 nasted함수임을 구분할 수 있게 하였다. 즉, 'nasted in 함수테이블'이 True이면 내부에 중첩함수를 가지고 있는 함수임을 알 수 있도록 했다.

그리고 run_user_func를 수행할 때, nasted를 확인하도록 했다. 만약 nasted라면 함수 내부에 있는 define문장을 따로 떼어내서 실행시키도록 했다. 가령 test case 20번 같은 경우 cube내부에 있는 sqrt를 define하는 부분을 run_expr하도록 했다. 그러면 내부 함수(sqrt)가 정의된다.

내부 함수에 대한 define을 수행했으므로, 원래 식에서 그 부분을 제거한다. 그러면 함수 내에서 전역 함수를 호출하는 것과 같은 형태의 lambda식이 된다. 따라서 lam을 호출해 처리한다.

nasted함수 호출이 종료되면 내부 함수를 제거해주어야 한다. 그렇지 않으면 전역 변수 테이블에 내부 함수가 계속 바인딩 되어 있다. 위의 예를 들면 cube를 호출하면 sqrt가 계속 바인딩 되어 있어서 cube호출 이전에는 sqrt를 단독으로 호출하지 못하지만 cube호출 이후에는 sqrt를 단독으로 호출할 수 있게 되는 것이다. 이를 방지하기 위해 run_user_expr에 nasted된 함수의 경우 함수 호출 종료시점에 내부 함수를 제거하는 코드를 포함했다.

2-2 결과 요약

테스트 케이스 번호	결과
1	>(define a 1) >a ...1
2	>(define b '(1 2 3) >b ...'(1 2 3)
3	>(define c (- 5 2) >c ...3

4	>(define d '(+ 2 3) >d ...'(+ 2 3)
5	>(define test b) >test ...'(1 2 3)
6	>(+ a 3) ...4
7	>(define a 2) >(* a 4) ...8
8	>((lambda (x) (* x -2)) 3) ...-6
9	>((lambda (x) (/ x 2)) a) ...1
10	>((lambda (x y) (* x y)) 3 5) ...15
11	>((lambda (x y) (* x y)) a 5) ...10
12	>(define plus1 (lambda (x) (+ x 1))) >(plus1 3) ...4
13	>(define mul1 (lambda (x) (* x a))) >(mul1 a) ...4
14	>(define plus2 (lambda (x) (+ (plus1 x) 1))) >(plus2 4) ...6
15	>(define plus3 (lambda (x) (+ (plus1 x) a))) >(plus3 a) ...5
16	>(define mul2 (lambda (x) (* (plus1 x) -2))) >(mul2 7) ...-16

17	>(define lastitem (lambda (ls) (cond ((null? (cdr ls)) (car ls)) (#T (lastitem (cdr ls)))))) >(lastitem '(1 2 5)) ...5
18	>(define square (lambda (x) (* x x))) >(define yourfunc (lambda (x func) (func x))) >(yourfunc 3 square) ...9
19	>(define square (lambda (x) (* x x))) >(define mul_two (lambda (x) (* 2 x))) >(define new_fun (lambda (fun1 fun2 x) (fun2 (fun1 x)))) >(new_fun square mul_two 10) ...200
20	>(define cube (lambda (n) (define sqrt (lambda (n) (* n n))) (* (sqrt n) n))) >(sqrt 4) There was an error getting the value of the variable sqrt is an undefined function. ...#F >(cube 4) ...64

2-3 장단점 분석

변수에 함수를 바인딩할 때, 전역 변수 테이블에 함수 테이블 자체를 만들어서 넣은 것이 장점이자 단점이라고 생각한다. 장점이라고 생각하는 이유는 무엇보다 구현할 때 편리했다. 함수 몸체, 매개변수가 테이블 형태로 고정되어 있기 때문에 actual parameter를 바인딩 할 때나, 함수의 내용을 가져와야 할 때 그냥 테이블을 참조하면 되었기 때문이다. 뿐만 아니라 Scope를 구현할 때에도 static parent의 함수 테이블을 참조하면 되는 방식이라 구현에 편리함이 있었다.

한편 이것을 단점이라고 생각하는 이유는 함수 테이블을 계속해서 유지하는 것이 메모리상의 낭비라고 생각한다. 단순히 함수의 내용만 가지고 있는 것이 아니라, 매개변수가 함수가 사용되지 않을 때에도 공간을 차지하고 있기 때문이다. 일반적인 유저 프로그램이 실행될 때처럼 함수가 실행될 때에만 매개변수를 위한 공간을 할당하도록 구현했다면 더 좋았을 것 같다.

덧붙여, 코드가 깔끔하지 않은 것도 단점인 것 같다. 구현을 하다가 특정 부분을 추가해야 할 부분이 생기면 다른 부분에 영향을 미칠까봐 if문을 많이 사용했는데, 그래서인지 if문이 너무 많이 사용된 것 같다.