Course Title	Programming Language Theory (ITP20005, Dec. 18 2020)			**Score**	___ / 31
Section	[✓ TF5] / TF6 (circle or underline yours)	**Student ID**	21701065	**Name**	Wonpyo Hong

I hereby testify before God and men that all of the answers are my own, and I have taken the exam to the best of my ability without resorting to unethical conduct.

Caution: Answer sheet without Student's signature is not valid. Signature _____홍원표______________ (write or type)

*** Answers written in Korean will not be marked. (Exam duration: 80 minutes)**
*** For students who are taking this by using your laptop, please use [File] >> [Make a copy] menu, and save it into your shared google directory for the PL class. (Like Quiz02)**
*** Open reference test but <u>do not blindly copy and paste</u>. <u>Write sentences based on your understanding in your own words.</u>**
*** While taking the exam online, you must turn on either webcam 'or' mic. This is just for symbolic action that we are taking the exam honestly together.**

1. Choose one topic or concept you think the most interesting one learned in this class. Then, discuss shortly why you choose it. (<u>6P</u> - Open question.)

Topic/Concept (<u>2P</u>):
I think recursion was the most interesting topic I've learned.
Discussion (Why?, <u>4P</u>):
Up until this time, I thought that recursion was a very natural concept that doesn't really need any external implementation. It was like opening a box inside a box like a russian doll or something. However, after learning in class, I've realized it can actually be implemented 'manually' when creating a new langauge.

2. For the concrete syntax of the AE language, we define two BNF grammars as follows.

Type 1	**Type 2**

<AE> ::= <num>
| {+ <OP>}
| {- <OP>}
<OP> ::= <num> <AE>
	<AE> ::= <num>
	| {+ <OP>}
	| {- <OP>}
	<OP> ::= <AE> <AE>

(1) Provide an example expression based on a BNF that is valid for both **Type 1** and **Type 2** (1P)

(+ 3 (+ 3 4))

(2) Provide an example expression that is valid only for **Type 2** (1P)

(+ (+ 3 4) (+ 4 3))

(3) Discuss the difference between **Type 1** and **Type 2** grammars in terms of expressiveness of the AE language? (2P)

Type 1 only allows number on the left hand side of the arithmetic expression. Type 2 allows both sides to have arithmetic expression within an arithmetic expression, which means it can make a better 'recursion' function.

3. Discuss about the relationship between an identifier and a variable (in what aspects are they similar and/or different?) (3P)

Variable is an inner term of identifier. Variable is like a name for a memory location that holds a value. Identifier is like a broader term of variable, or simply saying that it is a variable's name.

4. The following is a simple example of the continuation that can define a continuation for (+ ? 5). Update the code to create an continuation for an operator of '*', i.e., (+ (? 3 4) 5). (3P)

```
(define *k* #f)
(+ (call/cc                              ;; (+ (* 3 4) 5)
        (lambda (k)
          (begin
            (set! *k* k)
            (k (* 3 4)))))       5)
```

```
(define *a* #f)
(+ (call/cc
        (lambda (k)
          (begin
            (set! *d* k)
            (k (* 3 4)))))       5)
```

5. What is a meta-circular interpreter. Please find two example cases that show the meta-circularity of the interpreter implementation from the lectures. Explain why the examples you find show the meta-circularity.

- A meta-circular interpreter (1P):

It is an interpreter that is written on the same language, using only what's given, not needing any additional implementation, in order to add more features to the language or dialects.

- Two examples (2P):

Factorial w/ with → RCFAE

KCFAE, implementing withcc

- Why? (1P):

Whenever we update the interpreter, it is becoming a 'better' meta-circular interpreter, because we're 'adding' more stuffs to the previous language we've built.

*** Answer following questions based on the following LFAE implementation. (6~9)**

```
1:    #lang plai
2:
3:    (define-type LFAE
4:       [num    (n number?)]
5:       [add    (lhs LFAE?) (rhs LFAE?)]
6:       [sub    (lhs LFAE?) (rhs LFAE?)]
7:       [id     (name symbol?)]
8:       [fun    (param symbol?) (body LFAE?)]
9:       [app    (ftn LFAE?) (arg LFAE?)])
10:
11:   (define (parse sexp)
12:      (match sexp
13:          [(? number?)            (num sexp)]
14:          [(list '+ l r)          (add (parse l) (parse r))]
15:          [(list '- l r)          (sub (parse l) (parse r))]
16:          [(list 'with (list i v) e)  [redacted] ]
17:          [(? symbol?)            (id sexp)]
18:          [(list 'fun (list p) b)    (fun p (parse b))]
19:          [(list f a)             (app (parse f) (parse a))]
20:          [else                   (error 'parse "bad syntax: ~a" sexp)]))
21:
22:   (define-type DefrdSub
23:     [mtSub]
24:     [aSub (name symbol?) (value LFAE-Value?) (ds DefrdSub?)])
25:
26:   (define-type LFAE-Value
27:     [numV      (n number?)]
28:     [closureV   (param symbol?) (body LFAE?) (ds DefrdSub?)]
29:     [exprV      (expr LFAE?) (ds DefrdSub?)
30:                       (value (box/c (or/c false LFAE-Value?)))])
31:
32:   (define (strict v)
33:      (type-case LFAE-Value v
34:          [exprV (expr ds v-box)
35:                   (if (not (unbox v-box))
36:                        (local [(define v (strict (interp expr ds)))]
37:                           (begin (set-box! v-box v)
38:                                    v))
39:                        (unbox v-box))]
40:          [else v]))
41:   (define (num-op op)
42:        (lambda (x y)
43:            (numV (op (numV-n (strict x)) (numV-n (strict y))))))
44:
45:   (define num+ (num-op +))
46:   (define num- (num-op -))
47:
48:   (define (lookup name ds)
49:     (type-case DefrdSub ds
50:       [mtSub ()          (error 'lookup "free identifier")]
51:       [aSub  (i v saved) (if(symbol=? i name)
52:                             v
53:                             (lookup name saved))]))
54:
55:   (define (interp lfae ds)
56:     (type-case LFAE lfae
57:        [num (n)     (numV n)]
58:        [add (l r)     (num+ (interp l ds) (interp r ds))]
59:        [sub (l r)     (num- (interp l ds) (interp r ds))]
60:        [id  (s)       (lookup s ds)]
61:        [fun (p b)   (closureV p b ds)]
62:        [app (f a)   (local [(define f-val (strict (interp f ds)))
63:                               (define a-val (exprV a ds (box #f)))]
64:                           (interp (closureV-body f-val)
65:                                     (aSub (closureV-param f-val)
66:                                            a-val
67:                                            (closureV-ds f-val))))]))
```

6. (Line 67) The reason why we provide 'ds' (deferred substitution cache) of closureV instead of (mtSub) as in F1WAE supporting first-order functions is that function body also needs to substitute other identifiers except for the function parameter. (<u>1P</u>) ⇒ T | F

T

7. (Line 28) The purpose of the 'ClosureV' variant in LFAE-Value is to keep the function definition and the deferred substitution cache (ds) together for implementing dynamic scope. (<u>1P</u>) ⇒ T | F

F

8. In F1WAE, we defined the 'app' variant as follows: [app (ftn symbol?) (arg F1WAE?)]. The reason why we changed this as in Line 9 is we can call a function by both an identifier of a function definition in first class functions based on Lambda calculus (<u>1P</u>) ⇒ T | F

` T

9. Line 37 is executed at least once when we have an argument expression for a certain function (<u>1P</u>) ⇒ T | F

F

Your Student id: __21701065_____

10. Compare short-circuiting, deferred substitution, and laziness in various aspects. (3P)

Deferred Substitution is making the substitution delayed while laziness is making the evaluation delayed. Both are effective when making the interpreter compute faster.
Short Circuiting is stopping the evaluation expression right after the result is known, while laziness is using the evaluation when it is needed. Laziness doesn’t cut off like short circuiting, but delays.

11. Programming language is growing. Please discuss the following questions.

(1) Discuss the necessity that the programming language is growing (1P)

New technologies rise day by day, and new devices rise day by day. As there are many programming languages, their efficiency differs as well. It is inevitable that as new technologies rise, people require ‘better’ and more ‘efficient’ languages to emerge. Therefore, programming language is growing and has to grow in order to follow up the new technologies.

(2) Provide an example case that shows a language is growing. The example must be from one of your favorite programming languages. (1P)

Years ago when people tried to create a mobile application, they had to create using C or other old languages. However, nowadays there are many ways to create mobile applications. For example, flutter, using dart language, is a new language to accommodate both android and apple environments to be built in one language.

(3) For your preferred language, is there any new feature that can be supported in the future? Discuss in detail and why you need that feature of the language. (2P)

Since dart language is the most human-language-like programming language that I’ve encountered so far, it can probably go farther beyond to become a language that can be written in other human languages as well (not just in English).
This feature would become very needy in the future because as programming languages become easier and easier, there may be more and more people who want to try programming. However, there is a difficulty that people still have to learn English in order to learn programming languages.

- END -

In their hearts humans plan their course, but the Lord establishes their steps. (Proverbs 16:9)