Midterm Preparation

Regular Expression

- Syntax
  - Basic:
    - `ε`: represents an empty string, `ε` matches no characters string (empty string).
    - `a`: a single character, `a` matches a string containing only the character a.
  - Concatenation (sequencing): `RS` denotes the set of strings that can be obtained by concatenating a string in `R` and a string in `S`.
    - `RS = { αβ | α ∈ R, β ∈ S }`
  - Alternation: a vertical bar `|` separates alternatives.
    - `a|b` matches `a` or `b`.
  - Repetition:
    - `*`(Kleene star): the set of strings which are concatenations of zero or more occurrences of the preceding element.
      - `a*b` matches `b`, `ab`, `aab` and so on.
    - `+`: the set of strings which are concatenations of one or more occurrences of the preceding element.
      - `a+b` matches `ab`, `aab`, `aaab` and so on.
      - `a+` = `aa*`

Exercise

1. **[Easy]** Write an regular expression that matches the positive real number with the following restriction:

```
Should Match:
1.2
0.35
0.007
0.0
34.56
77.00
Should not match:
+1.2
-3.4
01.23
3
0
```

▶ Solution

```
([1-9][0-9]*|0)\.[0-9]+
```

2. **[Medium]** Write a regular expression to recognise patterns in the log files which contains *email id* and *date* separated by underscore.
    - <Email id>: email id for this question should have this format: `<name>@<domain>`:
        - `<name>`: it can be alphanumeric with two special characters. `.` and `-` are the only special characters allowed. They can occur multiple times and should be preceded and succeeded by atleast one alphanumeric character (That is, they could not appear consecutively, e.g. `--`, `..`, `-.`, or `.-`).
        - `<domain>`: it should contain alphanumeric characters with only one `.` in between.
    - <Date>: the date can be in `mm-dd-yyyy` or `yyyy-mm-dd` format with the following rules:
        - `mm`: should between `01` to `12`.
        - `dd`: should betweeen `01` to `30` if the `mm` is even. Otherwise, `dd` should between `01` to `31`.

- For example, the following string should be accepted:

```
john.wick2-cs.nyu@abc.com_2020-01-01
ROBERT.Smith@example.com_03-12-2008
test-one.one-test@123.123_0000-01-01
```

- Note: This question is created by Goutham Panneeru (gp1521@nyu.edu).

▶ Solution

```
name   := [a-zA-Z0-9]+((\.|\-)[a-zA-Z0-9]+)*
domain := [a-zA-Z0-9]+\.[a-zA-Z0-9]+
mmdd   :=
((01|03|05|07|09|11)-(0[1-9]|[1-2][0-9]|3[0-1]))|((02|04|06|08|10|12)-
(0[1-9]|[1-2][0-9]|30))
yyyy   := \d\d\d\d
Thus, the result should be:
    name\@domain\_((mmdd\-yyyy)|(yyyy\-mmdd))
```

Context Free Grammar

- Terminals: the set of the alphabet of the language
- Nonterminals: the set of variables, each variable represents a different type of phrase or clause in the sentence
- Productions: rules for replacing a single non-terminal with a string of terminals and non-terminals
- Starting symbol: a nonterminal, used to represent the whole sentence (or program)

Exercise

Provide a context free grammar over the alphabet {`a`,`b`} such as:

1. **[Easy]** Accept a string that `a` followed by `b` and the number of `a`'s is more than the number of `b`'s:

```
a
aab
aaab
aaaaabbb
...
```

▶ Solution

```
S -> aA
A -> aA  | B
B -> aBb | ε
```

2. **[Hard]** Challenge yourself to consider all strings with more a's than b's:

```
a
bbabaaa
ababaab
aba
baaaa
...
```

▶ Solution

```
S -> aM  | MS  | aS
M -> aMb | bMa | MM | ε
```

- Here is the website for testing the correctness of CFG.

Static vs. Dynamic Scoping

1. Static scoping: binding of a name is determined by rules that refer only to the program text. (i.e. its syntactic structure)
2. Dynamic scoping: binding of a name is given by the most recent declaration encountered during run-time.

Exercise

Consider this code snippet:

```
int a = 0, b = 0, c = 0; // Assume global variables
void q(); // Declare function q

```

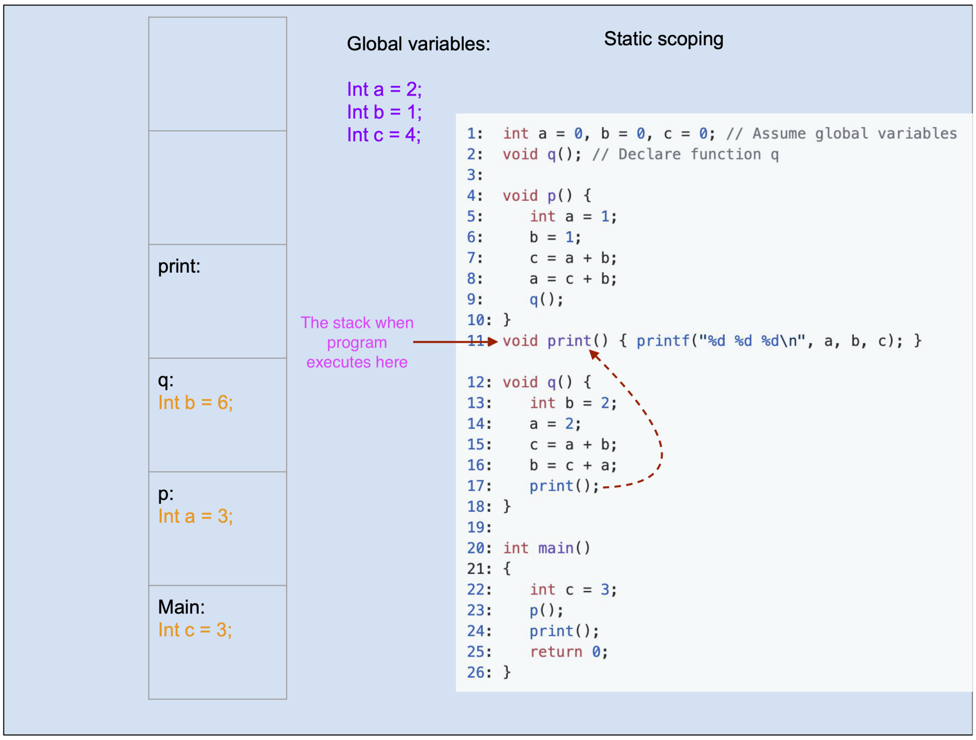
```
void p() {
   int a = 1;
   b = 1;
   c = a + b;
   a = c + b;
   q();
}
void print() { printf("%d %d %d\n", a, b, c); }

void q() {
   int b = 2;
   a = 2;
   c = a + b;
   b = c + a;
   print();
}
int main()
{
   int c = 3;
   p();
   print();
   return 0;
}
```

1. **[Medium]** In c programming, we know that c is using static scoping. What does this program print when it runs?

▶ Solution

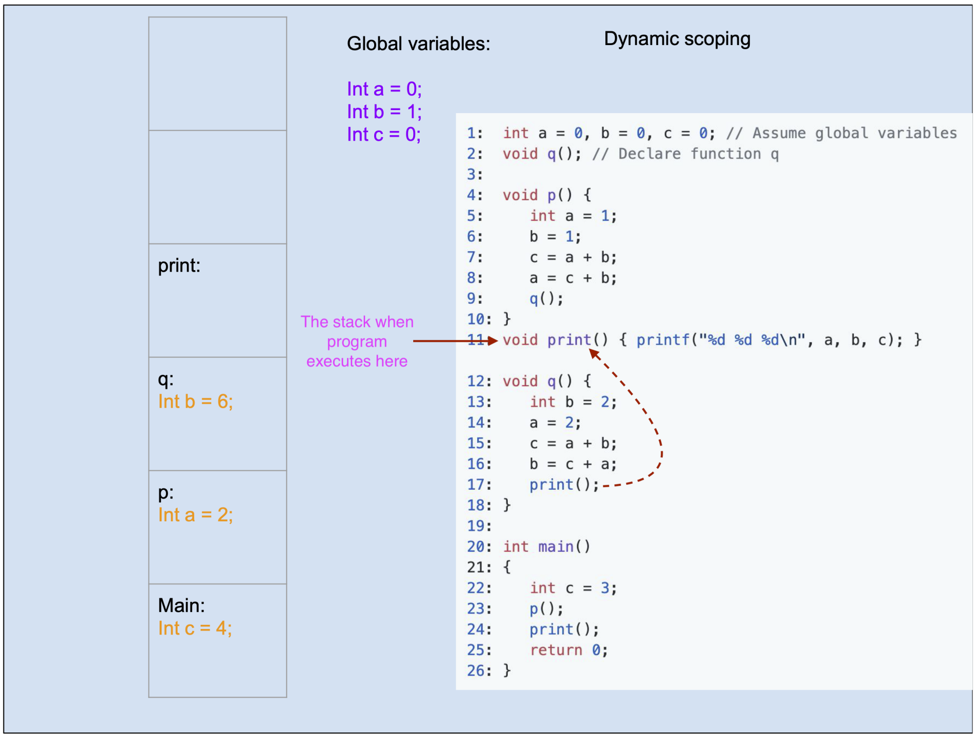
```
2 1 4
2 1 4
```

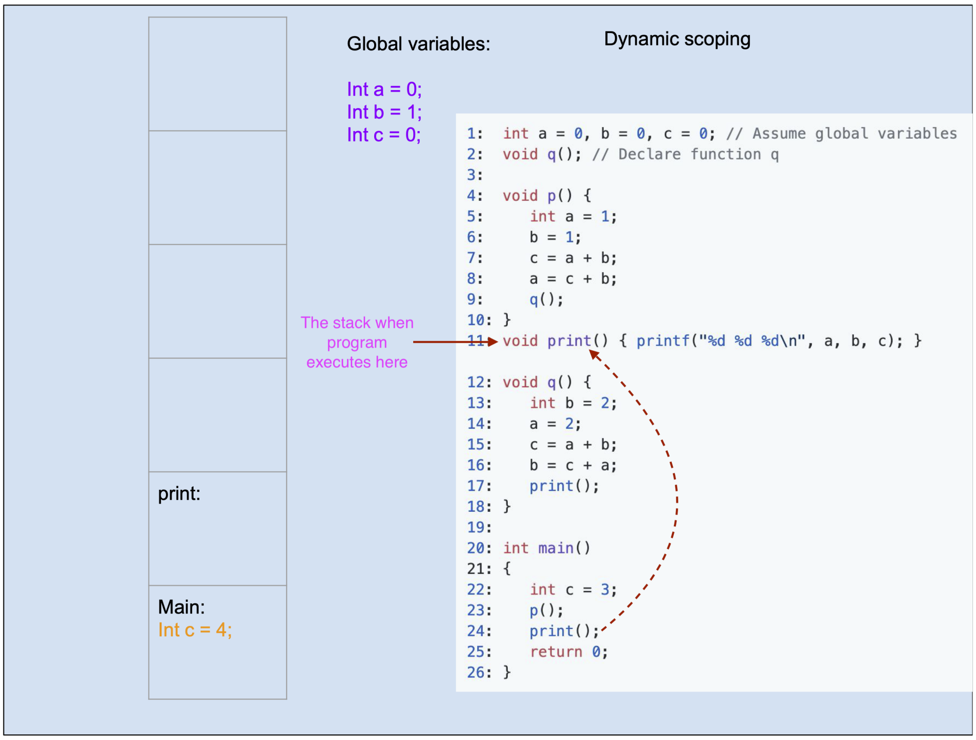


2. **[Hard]** Now assume the program is running under dynamic scoping. What does this program print?

▶ Solution

```
2 6 4
0 1 4
```





- To consider dynamic scoping, you can use **stack frame** to remeber the declaration order during the running time. For instance, here is a simple stack frame when the program called `print()` inside function `q()`:

```
|                          |
|==========================|
| print():                 |
|==========================|
| q():                     |                                  global var
|   int b = 6              |                                 |  a = 0  |
|==========================|                                 |  b = 1  |
| p():                     |                                 |  c = 0  |
|   int a = 2              |
|==========================|
| main():                  |
|   int c = 4              |
|==========================|
| ...                      |
```

- The variable `a`, `b` and `c` inside `print()` should be bounded the most recent declaration, where referred `a` is bounded by variable `a` inside function `p()`, `b` is bounded by variable `b` inside function `q()`, and `c` is bounded by variable `c` inside function `main()`.

Parameter Passing Modes

1. Strict evaluation: call-by-value, call-by-reference
2. Lazy evaluation: call-by-name, call-by-need

Exercise

Consider this following Pseudo code:

```
/* static scoping */
Int Incr(Int& k) { // pass by reference
    k = k + 1
    return k
}

Int z = 1
/*Note: evaluations for addition and printf are both from left to right*/

Void F(Int x, Int y) { // Suppose formal could be assigned
    x = y + z;
    Printf("%d %d\n", x, y);
}

F(z, Incr(z));
Printf("%d\n", z);
```

What does this program print if we make the following assumptions about the parameter passing modes for the parameters x and y of f:

1. **[Easy]** x and y using call-by-value parameter

▶ Solution

```
4 2
2
```

```
/* static scoping */
Int Incr(Int& k) { // pass by reference
    k = k + 1
    return k=z
}

Int z = 1
/*Note: evaluations for addition and printf are both from left to right*/
                1         2
                ||        ||
Void F(Int x, Int y) { // Suppose formal could be assigned
  4 = x = y + z;
          2   2
    Printf("%d %d\n", x, y);
            4   2
}

F(z, Incr(z));
Printf("%d\n", z); z=2
```

2. **[Easy]** x is call-by-reference and y is call-by-value

▶ Solution

```
4 2
4
```

```
/* static scoping */
Int Incr(Int& k) { // pass by reference
    k = k + 1
    return k =z
}

Int z = 1
/*Note: evaluations for addition and printf are both from left to right*/
              z        2
              ||       ||
Void F(Int x, Int y) { // Suppose formal could be assigned
            2   2
4=z<=>x = y + z;
    Printf("%d %d\n", x, y);
}             4    2

F(z, Incr(z));
Printf("%d\n", z); z=4
```

3. **[Medium]** x is call-by-value and y is call-by-name

▶ Solution

```
4 3
3
```

```
/* static scoping */
Int Incr(Int& k) { // pass by reference
    k = k + 1
    return k =z
}

Int z = 1
/*Note: evaluations for addition and printf are both from left to right*/
              1     Incr(z)
              ||       ||
Void F(Int x, Int y) { // Suppose formal could be assigned
         Incr(z)=2  2
  4 = x = y + z;
    Printf("%d %d\n", x, y);
}            4    ||
                Incr(z) = 3

F(z, Incr(z));
Printf("%d\n", z); z=3
```

4. **[Medium]** x is call-by-reference and y is call-by-name

▶ Solution

```
4 5
5
```

```
/* static scoping */
Int Incr(Int& k) { // pass by reference
    k = k + 1
    return k = z
}

Int z = 1
/*Note: evaluations for addition and printf are both from left to right*/
                z         Incr(z)
                ||          ||
Void F(Int x, Int y) { // Suppose formal could be assigned
           Incr(z) = 2
              ||      2
4 = z = x = y + z;
    Printf("%d %d\n", x, y);
}                4   ||
                 Incr(z) = 5

F(z, Incr(z));
Printf("%d\n", z); z = 5
```

Lambda Calculus

Exercise

1. **[Easy]** Determine the set of free variables inside this lambda expression:

```
(λ x. (λ y. x) y (λ x. x)) (λ z. z) x
```

▶

```
free variable: y, x
```

2. Consider the church encoding, we know that:

```
true = (λ x y. x)
false = (λ x y. y)
0 = (λ s z. z)
1 = (λ s z. s z)
succ = (λ n s z. s (n s z))
pair = (λ x y b. b x y)
fst = (λ p. p true)
snd = (λ p. p false)
pred = λ n. snd (n (λ p. pair (succ (fst p)) (fst p)) (pair 0 0))
```

[Hard] How do we compute `pred 1` to get `0` via beta reduction?

▶ Solution

```
    pred 1      #| By mixed order |#
=> (λ n. snd (n (λ p. pair (succ (fst p)) (fst p)) (pair 0 0))) 1   ; by
def of pred
=> snd (1 (λ p. pair (succ (fst p)) (fst p)) (pair 0 0))            ; do
one step for λ n
=> snd ((λ s z. s z) (λ p. pair (succ (fst p)) (fst p)) (pair 0 0)) ; by
def of 1
=> snd ((λ p. pair (succ (fst p)) (fst p)) (pair 0 0))              ; do
two steps for λ s and λ z
=> snd (pair (succ (fst (pair 0 0))) (fst (pair 0 0)))              ; do
one step for λ p
=> snd (pair (succ (fst (pair 0 0))) (fst ((λ x y b. b x y) 0 0)))  ; by
def of pair
=> snd (pair (succ (fst (pair 0 0))) (fst (λ b. b 0 0)))            ; do
two steps for λ x and λ y
=> snd (pair (succ (fst (pair 0 0))) ((λ p. p true) (λ b. b 0 0)))  ; by
def of fst
=> snd (pair (succ (fst (pair 0 0))) ((λ b. b 0 0) true))           ; do
one step for λ p
=> snd (pair (succ (fst (pair 0 0))) (true 0 0))                    ; do
one step for λ b
=> snd (pair (succ (fst (pair 0 0))) ((λ x y. x) 0 0))              ; by
def of true
=> snd (pair (succ (fst (pair 0 0))) 0)                             ; do
two steps for λ x and λ y
=> (λ p. p false) (pair (succ (fst (pair 0 0))) 0)                  ; by
def of snd
=> (pair (succ (fst (pair 0 0))) 0) false                           ; do
one step for λ p
=> ((λ x y b. b x y) (succ (fst (pair 0 0))) 0) false               ; by
```

```
def of pair
=> false (succ (fst (pair 0 0))) 0                              ; do
three steps for λ x, λ y and λ b
=> (λ x y. y) (succ (fst (pair 0 0))) 0                         ; by
def of false
=> 0                                                            ; do
two steps for λ x and λ y
```

Scheme Programming

Exercise

1. **[Medium]** `pack`: define a function `pack` that packs consecutive duplicates of list elements into sublists.

For example:

```
> (pack '(a a a a b c c a a d e e e e))
((a a a a) (b) (c c) (a a) (d) (e e e e))
```

- Intuition: using foldr will help you simplify the conversion. Basically, foldr will iterate the list from end to begin and use the input function `f` with two arguments to reduce the result (an element on the list and single value `z`). Thus, you can create an empty list as `z` for calling `foldr`. During `foldr` iterating list, check element in the list and contruct either current element should build a new sublist or append it into the first sublist inside `z`. For example, consider giving `foldr` function a list `'(a a a b b)`:

```
         f =>    '((a a a) (b b))
        / \
       a    f =>  '((a a) (b b))
           / \
          a    f => '((a) (b b))
        / \
       a    f => '((b b))
           / \
          b    f => '((b))
              / \
             b    z = '()
```

▶ Solution

```
; foldr
(define (foldr f s L)
(if (null? L) s (f (car L) (foldr f s (cdr L)))))

; pack
(define (pack ls)
  (cond
```

```
      ((null? ls) '())
      (else (foldr (lambda (x z)
        (cond
            ((null? z) (cons (cons x '()) z))
            ((= x (caar z)) (cons (cons x (car z)) (cdr z)))
            (else (cons (cons x '()) z))
        )) '() ls)
      )
    )
  )
```

2. **[Medium]** `split`: define a function `split` that splits an input list into two parts by given a length of the first part. For instance:

```
> (split '(a b c d e f g) 3)
((a b c)(d e f g))
```

▶ Solution

```
; rev
(define (rev ls)
  (letrec
    ((rev_acc (lambda (acc rv)
       (if (null? acc) rv
     (rev_acc (cdr acc) (cons (car acc) rv))))))
     (rev_acc ls '()))
)

; split
(define (split ls n)
  (letrec ((split-rec (lambda (ls n res)
      (cond
    ((= n 0) (cons (rev (car res)) (cons ls '())))
    (else (split-rec (cdr ls) (- n 1) (cons (cons (car ls) (car res))
(cdr res)))))
  )))
  (split-rec ls n '(())))
)
```