Universidad Nacional Autónoma de México
Facultad de Ciencias
Compiladores
Tarea 3
Laura Itzel Rodríguez Dimayuga

Ejercicio 1

Indica los valores asignados a w, x, y y z. en los siguientes dos códigos estructurados por bloques. Muestrala tabla de símbolos en cada bloque con una implementación imperativa en cada caso:

```
int w,x,y,z;
int i = 4; int j = 5;
{
    int j =7;
    i =6;
    w = i+j;
}
x = i+j;
{
    int i = 8;
    y = i+j;
}
z = i+j;
```

```
int w,x,y,z;
int i = 3; int j = 4;
{
    int i = 5;
    w = i+j;
}
x = i+j;
{
    int j = 6;
    i = 7;
    y = i+j;
}
z = i+j;
```

Solución.

Variable	Scope	Comentario
int i =4	Bloque 1	Nueva variable
int j = 5	Bloque 1	Nueva variable
int j = 7	Bloque 2	Nueva variable, solo queda en el Bloque 2
i=6	Bloque 2	Afecta al Bloque 1
x = 9	Bloque 1	Afecta al Bloque 1
int i =8	Bloque 3	Nueva variable, solo queda en el Bloque 3
y = 13	Bloque 3	Afecta al Bloque 1
z = 6 + 5	Bloque 1	El valor de i cambio, pero el de j es el del primer reglon
Final		w = 13, x = 9, y = 13, z = 11

■

Ejercicio 2

Divide el siguiente programa en C++ en lexemas y genera los tokens correspondientes:

```
    float limitedSquare(x) float x; {
        /* return x-squared, bit never mover than 100 */
        return(x <= -10.0 || x>=10.0) ? 100 : x*x;
    }
```

Solución.

Después del escaneo no tenemos comentarios. Ni espacios en blanco. Lexemas: `float, limitedSquare, (, x, ), float, x, ;,, return, (, x, <=, -10.0, ||, x, >=, 10.0, ), ?, 100, :, x, *, x, ;,`

Tokens:

```
<float>
<id, limitedSquare>
<(>
<id, x>
<)>
<float>
<id, x>
<;>
<{>
<return>
<(>
<id, x>
<<=>
<float,-10.0>
<||>
<id, x>
<>=>
<float,10.0>
<)>
<?>
<100>
<:>
<id, x>
<*>
<id, x>
<;>
<}>
```

■

Ejercicio 3

Define una función recursiva que compute los prefijos de una expresión regular. La base de tal función recursiva es:

$$prefix(\varepsilon) = \{\varepsilon\} prefix(a) = \{a\}$$

Completa la definición.

Solución. Para definir los prefijos de una expresión regular, podemos considerar las siguientes reglas:

Podemos definir las reglas que aplicamos para cada una de las operaciones básicas de las expresiones regulares:

$$prefix(ab) = \{a, ab\} prefix(a|b) = \{a, b, a|b\} prefix(a*) = \{a^n | n \geq 0\}$$

■

Ejercicio 4

Sea $p = 1217$, $\alpha = 3$ y $\beta = 37$. Calcula $log_\alpha \beta$ mediante el algoritmo de Calculo de indices.

Solución.

Queremos calcular $log_3 37 \mod 1217$. Para esto ocupe el siguiente codigo. Aqui deifinimos las llaves

```python
import random
from math import gcd, isqrt
from typing import List, Optional, Tuple

# Function to factorize a number n using the factor base
def factorize(n: int, factor_base: List[int]) -> Optional[List[int]]:
    result = [0] * len(factor_base)  # Initialize exponents counter for each prime in the base
    temp = n  # Copy of n to factorize

```

```python
    for i, prime in enumerate(factor_base):
        while temp % prime == 0:
            result[i] += 1  # Increment the exponent for the prime
            temp //= prime  # Divide n by the prime

    if temp == 1:  # If we've completely factorized n
        return result
    return None  # If not fully factorizable, return None

# Function to calculate the modular inverse of a number a modulo m
def mod_inverse(a: int, m: int) -> int:
    def extended_gcd(a: int, b: int) -> Tuple[int, int, int]:
        if a == 0:
            return b, 0, 1
        gcd, x1, y1 = extended_gcd(b % a, a)
        x = y1 - (b // a) * x1
        y = x1
        return gcd, x, y

# Step 1: Find a factor base
def find_factor_base(p: int) -> List[int]:
    def is_prime(n: int) -> bool:
        if n < 2:
            return False
        for i in range(2, isqrt(n) + 1):
            if n % i == 0:
                return False
        return True

    limit = max(int(pow(p.bit_length() * 0.693, 1.414)), 20)
    factor_base = [n for n in range(2, limit + 1) if is_prime(n)]
    max_base_size = min(20, p.bit_length() // 2)
    return factor_base[:max_base_size]

# Step 2: Find relations
def find_relations(alpha: int, p: int, factor_base: List[int], verbose: bool = True) -> List
    [Tuple[List[int], int]]:
    relations = []
    exponents_used = set()
    max_attempts = 5000
    attempts = 0

    while len(relations) < len(factor_base) + 5 and attempts < max_attempts:
        k = random.randrange(1, p.bit_length() * 100)
        if k in exponents_used:
            continue
        exponents_used.add(k)

        power = pow(alpha, k, p)
        exponents = factorize(power, factor_base)

        if exponents is not None:
            relations.append((exponents, k))
            if verbose:
                factorization = ' * '.join([f"{factor_base[i]}^{e}" for i, e in enumerate(
    exponents) if e > 0])
                print(f"Relation {len(relations)}: {alpha}^{k} mod {p} = {power}, {
    factorization}")

        attempts += 1

    if len(relations) < len(factor_base) + 1:
        raise ValueError(f"Insufficient relations found after {max_attempts} attempts.")

    return relations

# Funcion para resolver un sistema de ecuaciones lineales mod p-1
def solve_linear_system(relations: List[Tuple[List[int], int]], p: int) -> Optional[List[int
```

```
    ]]:
    n = len(relations[0][0])  # Numero de incognitas
    m = len(relations)  # Numero de ecuaciones
    matrix = [[x for x in eq[0]] + [eq[1]] for eq in relations]  # Construimos la matriz aumentada

    # Realizamos una eliminacion gaussiana para resolver el sistema
    for i in range(n):
        pivot_row = -1
        min_val = float('inf')
        for j in range(i, m):
            if 0 < abs(matrix[j][i]) < min_val:
                min_val = abs(matrix[j][i])
                pivot_row = j

        if pivot_row == -1:
            continue  # Si no hay pivote, continuamos con la siguiente columna

        # Intercambiamos filas si es necesario
        if pivot_row != i:
            matrix[i], matrix[pivot_row] = matrix[pivot_row], matrix[i]

        pivot = matrix[i][i]
        try:
            pivot_inv = mod_inverse(pivot, p-1)  # Calculamos el inverso del pivote mod (p-1)
            # Hacemos una eliminacion hacia atras
            for j in range(i, n + 1):
                matrix[i][j] = (matrix[i][j] * pivot_inv) % (p-1)

            for k in range(m):
                if k != i and matrix[k][i] != 0:
                    factor = matrix[k][i]
                    for j in range(i, n + 1):
                        matrix[k][j] = (matrix[k][j] - factor * matrix[i][j]) % (p-1)
        except ValueError:
            continue

    # Comprobamos si el sistema tiene una solucion unica
    rank = sum(1 for row in matrix[:n] if any(x != 0 for x in row[:-1]))
    if rank < n:
        return None  # Si el rango es menor que el numero de incognitas, el sistema no tiene solucion unica

    return [row[-1] for row in matrix[:n]]  # Devolvemos la solucion del sistema

#Step 3: Compute the discrete logs
def compute_discrete_log(p: int, alpha: int, beta: int, factor_base: List[int],relations) -> Optional[int]:
    discrete_logs = None
    # Intentamos resolver el sistema de ecuaciones lineales
    for i in range(min(5, len(relations) - len(factor_base))):
        try:
            candidate_logs = solve_linear_system(relations[i:t+i], p)
            if candidate_logs is not None:
                discrete_logs = candidate_logs
                break
        except Exception:
            continue

    return discrete_logs


# Step 4: Compute the discrete logarithm
def calculate(p: int, alpha: int, beta: int, factor_base: List[int], discrete_logs: List[int]) -> Optional[int]:
    max_attempts = 5000
    attempts = 0
```

```
137
138     while attempts < max_attempts:
139         k = random.randrange(1, p.bit_length() * 100)
140         gamma = (beta * pow(alpha, k, p)) % p
141         exponents = factorize(gamma, factor_base)
142
143         if exponents is not None:
144             log_sum = sum(e * l for e, l in zip(exponents, discrete_logs)) % (p - 1)
145             result = (log_sum - k) % (p - 1)
146
147             if pow(alpha, result, p) == beta:
148                 return result
149
150         attempts += 1
151
152     raise ValueError("Failed to compute discrete logarithm.")
153
154 # Main function
155 def index_calculus(p: int, alpha: int, beta: int, verbose: bool = True) -> Optional[int]:
156     factor_base = find_factor_base(p)
157     relations = find_relations(alpha, p, factor_base, verbose)
158     discrete_logs = compute_discrete_log(p, alpha, beta, factor_base, relations)
159     return calculate(p, alpha, beta, factor_base, discrete_logs)
160
161 # Testing block
162 if __name__ == "__main__":
163     p = 1217
164     alpha = 3
165     beta = 37
166     try:
167         result = index_calculus(p, alpha, beta)
168         print(f"\nResult: log_{alpha}({beta}) = {result} (mod {p-1})")
169     except Exception as e:
170         print(f"\nError: {e}")
171
172 # Resultado: log_3(37) = 588 (mod 1216)
```

Quiero aclarar que cuando lo corri no puse hacer que la eliminacion gaussiana siempre funcionara. ■

Ejercicio 5

Realiza una breve investigación acerca del esquema de firma de Merkle(MSS). La investigación debe responder como mínimo las siguientes preguntas:

- ¿Qué es un árbol de Merkle?
- ¿De qué manera se generan firmas?
- ¿De qué manera se verifican las firmas?

Solución.

- **Árbol de Merkle:** Se puede considar como una estructura de datos en donde cada hoja es un hash de un bloque de datos y cada nodo interno es un hash de las concatenaciones de los nodos hijos. La raíz del árbol es el hash de la concatenación de las hojas. Nacieron en 1980 con Ralph Merkle.
- **Generación de firmas:** Hacemos un hash sobre todas las hojas del árbol(que pueden ser las palabras que queremos cifrar).
- **Verificación de firmas:** Se puede verificar la firma de manera eficiente utilizando el hash de la raíz del árbol y el hash de las hojas.

  Aqui esta un ejemplo del algoritmo de firma de Merkle:

```python
import hashlib

# Referencia: https://pt.w3d.community/jennyt/arbol-de-merkle-todo-lo-que-necesitas-saber-5ak5
# Funcion para calcular el hash de un dato
def calcular_hash(dato):
    return hashlib.sha256(dato.encode()).hexdigest()

# Funcion para construir el arbol de Merkle
def construir_arbol_merkle(datos):
    # Calculamos los hashes de las hojas
    hojas = [calcular_hash(dato) for dato in datos]

    # Construimos el arbol de manera recursiva
    while len(hojas) > 1:
        # Agrupamos los hashes en pares y los combinamos
        hojas = [calcular_hash(hojas[i] + hojas[i+1]) for i in range(0, len(hojas), 2)]

    # Devolvemos la raiz del arbol
    return hojas[0]

# Verificar la integridad de los datos
def verificar_integridad(datos, raiz_merkle, dato_modificado):
    # Calculamos el hash del dato modificado
    hash_modificado = calcular_hash(dato_modificado)

    # Si el hash del dato modificado es igual a la raiz del arbol de Merkle, los datos estan integros
    if hash_modificado == raiz_merkle:
        print("Los datos estan integros.")
    else:
        print("Los datos han sido modificados.")

# Ejemplo de uso
datos = ["Bitcoin", "Ethereum", "Litecoin", "Monero", "Hyperledger", "Corda", "Multichain"]
raiz_merkle = construir_arbol_merkle(datos)

print("Raiz del arbol de Merkle:", raiz_merkle)

# Supongamos que modificamos un dato
dato_modificado = "Bitcoin"
verificar_integridad(datos, raiz_merkle, dato_modificado)
```

■