



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Práctica 1

Conversión de Expresiones Regulares a NFA.

ALUMNO

Gallegos Cortes José Antonio - 320316566

PROFESOR

Victor Germán Mijangos.

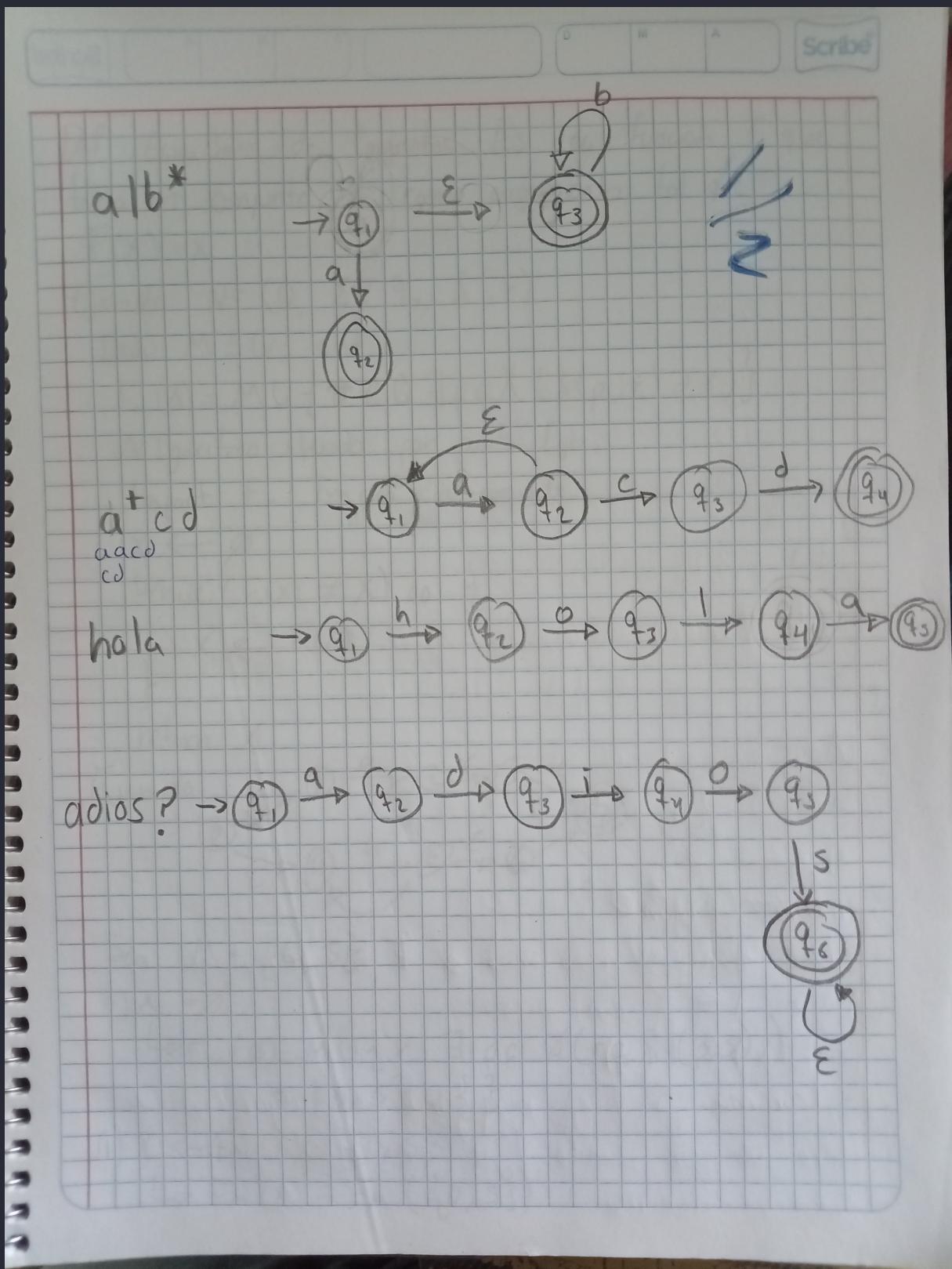
AYUDANTES

Adrián Martínez Manzo
Teresa Becerril Torres

ASIGNATURA

Compiladores.

Validación.



Descripción de la implementación.

Estructuras de datos utilizadas

Para la implementación del conversor de expresiones regulares a NFA, utilizamos las siguientes estructuras de datos:

En regex.h:

- **token**: Esta estructura representa un token de la expresión regular, contiene un carácter (**value**) y su tipo (**type**) que puede ser TOKEN_CHAR, TOKEN_DOT, TOKEN_STAR, TOKEN_PLUS, TOKEN_QUESTION, TOKEN_PIPE.
- **regex**: Esta estructura representa una expresión regular como un arreglo dinámico de tokens, debe de ser dinamico porque al agregar las concatenaciones explicitas va a crecer. Mantiene un puntero a los **items**, el tamaño actual **size** y la capacidad del arreglo **capacity**.
- **stack**: Esta es una pila como las de siempre, la implementamos con un arreglo dinámico de caracteres para de igual forma hacer que crezca, aunque por lo que he visto, no es tan necesario porque las expresiones a lo más pueden crecer (casi) el doble cuando se les pone concatenación, por ejemplo abc puede crecer a **a.b.c**, esta pila la utilizamos para el algoritmo Shunting Yard.

En nfa.h:

- **transition**: Representa una transición del autómata. Contiene el símbolo **symbol**, un indicador de si es transición **is_epsilon**, el estado destino **to** y un puntero a la siguiente transición **next** para formar una lista enlazada, tiene que ser una lista enlazada porque necesitamos poder 'soportar' o representar varias transiciones a la vez, recordemos que un estado puede tener más de una transición.
- **state**: Representa un estado del NFA. Contiene un **id** y una lista enlazada de transiciones.
- **nfa**: Representa el autómata ya completo. Contiene punteros al estado inicial que tenemos como **start** y al estado de aceptación que tenemos como **accept**.
- **fragment**: Representa un fragmento de NFA durante la construcción, Representa un "sub-autómata" durante la construcción mediante el algoritmo de Thompson, no vamos a tener el automata completo de golpe, vamos a tener fragmentos con estado inicial y estado de aceptación, vamos a poder combinar fragmentos conforme se procesan los operadores de la regex.

Algoritmos implementados en regex.c

1. Detección de caracteres normales

La función **is_normal_char(char t)** nos ayuda a saber si un carácter pertenece a los simbolos especiales o si es un carácter cualquiera:

```
int is_normal_char(char t){  
    const char *symbols = ".|*+?()";  
    for (int i = 0; symbols[i] != '\0'; i++) {  
        if (t == symbols[i]) return 0;  
    }  
    return 1;  
}
```

Primero recibimos un carácter **t** (es decir a la función ya debe de llegar el carácter separado). Tomamos ese carácter y lo vamos comparando con el arreglo de caracteres especiales, para evitar llenar de if's o de or's mejor declaré ese arreglo con los simblos, lo recorre hasta el final y compara el carácter con cada uno de los simblos, en caso de que coincida regresamos 0 porque querría decir que

es un simbolo especial, en caso de cualquier otra cosa regresamos 1 porque sería un caracter normal.

```
[admin@MacBook-Air-de-admin] - [~/Documents/personal/Escuela/Compiladores/practicas/regex_to_nfa/src] - [sáb. feb 28, 21:55]
└─[$] <git:(development*)> gcc -DTEST regex.c -o test && ./test
Test mode:
1 <char>      - test is_normal_char
2 <char1><char2> - test needs_concatenation
3 <regex>       - test insert_concatenation
4 <regex>       - test infix_to_postfix
q               - quit

4 a*b+s
infix_to_postfix("a*b+s") = "ab*s+"
1 a
is_normal_char('a') = 1
1*
is_normal_char('*') = 0
5
```

2. Detección de concatenación implícita

La función `needs_concatenation(char c1, char c2)` nos dice si es necesario insertar un operador de concatenación explícito:

```
int needs_concatenation(char c1, char c2){
    int c1_ends = is_normal_char(c1) || c1 == ')' || c1 == '*' || c1 == '+'
    || c1 == '?';
    int c2_starts = is_normal_char(c2) || c2 == '(';
    if (c1 == ')' || c2 == '(') return 0;
    return c1_ends && c2_starts;
}
```

La idea aquí surge de ver en qué casos es necesaria una concatenación `'.'`. Podemos ver que las concatenaciones solo son necesarias después de una expresión completa y antes del comienzo de otra expresión. Al decir 'expresión completa' me refiero a una subexpresión válida, los caracteres como `+, *, ?, ?` nos ayudan a terminar de formar una subexpresión, digamos que la completa y ayuda a ser válida. Nosotros queremos unir dos expresiones bien formadas, podemos hacerlo solo cuando ocurren estas dos cosas:

- El primer simbolo puede completar/finalizar una expresión válida. es decir cuandot tenemos un `+, *, ?, ?` etc o cuando tenemos un carácter.
- Y cuando el segundo simbolo puede iniciar una sub-expresión válida.

3. Inserción de concatenaciones

La función `insert_concatenation(const char *str)` inserta `.` donde sea necesario:

```
char* insert_concatenation(const char *str){
    int length = strlen(str);
    char *new_str = malloc(2*length + 1);
    int j = 0;
    for (int i = 0; i < length; i++) {
        new_str[j++] = str[i];
        if (i < length - 1 && needs_concatenation(str[i], str[i + 1])) {
            new_str[j++] = '.';
        }
    }
    new_str[j] = '\0';
    return new_str;
}
```

Esta es sencilla, vamos a recibir la expresión original y para no modificarla y perder el valor original vamos a construir una segunda expresión con las concatenaciones agregadas, pero es muy importante hacer la copia más grande que la longitud de la expresión original, porque al insertar la concatenación explícita vamos a aumentar el tamaño de la expresión (además de que modificar la expresión 'al vuelo' a veces complica todo de más). Lo aumenté a dos veces el tamaño original porque a lo más tendríamos eso, en el peor caso vamos a insertar $2n-1$ a la expresión. Por ejemplo la expresión 'aaa' de longitud 3 la vamos a transformar en a.a.a de longitud 5, es decir, el crecimiento máximo de la cadena ocurre cuando necesitamos concatenación en todos los pares consecutivos de la cadena, no es posible que crezca más de $2\text{length}-1$, pero para tener un pequeño margen lo dejé en $2\text{length}+1$.

Para construir la expresión con concatenaciones vamos a recorrer la expresión original comparando de 2 en 2. Esto porque debemos de saber si 'a' necesita ser concatenado con 'b', y para ello vamos a usar la expresión que construimos anteriormente, la que nos dice si necesitan ser concatenados esos dos caracteres, en caso de que la función nos regrese un 1, simplemente agregamos una concatenación y seguimos.

En resumen, creamos la copia vacía del doble del tamaño de la cadena original, vamos a agregar el elemento i a el lugar j de la cadena original, entramos al for, preguntamos si necesita una concatenación, en caso de que si, agregamos la concatenación al j+1, en caso de que no simplemente regresamos al for, agregamos el siguiente elemento de la cadena y repetimos.

4. Algoritmo Shunting Yard

La función `infix_to_postfix(const char *str)` convierte la expresión de notación infija a postfixa:

1. Si es un carácter, lo agregamos directamente al output.
2. Si es '(', hacemos push a la pila.
3. Si es ')', hacemos pop hasta encontrar el '. Aquí hay algo importante, debemos de ser capaces de detectar una expresión con parentesis mal balanceados porque en caso de que no lo hagamos, vamos a hacer pop hasta el final de la pila y tendríamos un error.
4. Si es un operador, comparamos la precedencia de los operadores y hacemos pop de los que tengan mayor o igual precedencia, luego hacemos push.
5. Al final, vaciamos la pila al output.

Aquí en general es eso, en el código se ve bastante más de lo que en realidad es. Primero hacemos lo mismo que en la anterior, vamos a crear una copia vacía de la cadena pero en este caso la cadena va a tener las concatenaciones explícitas, vamos a llamar a la función `insert_concatenation`. Vamos a recorrer la cadena comparando carácter por carácter, usamos la función auxiliar `is_normal_char` para detectar si es un carácter normal, en caso de que lo sea simplemente vamos a agregarlo al `output`, que en este caso es la cadena `postfix`. En caso de que no sea un `normal char`, debería de ser un símbolo especial, pero antes de verificar eso debemos de ver el caso especial de este algoritmo, los casos de los paréntesis, este caso es simple, si detectamos un '(' lo agregamos al `stack` y continuamos de manera normal, en cuanto encontramos el ')' vamos a sacar elementos del `stack` hasta encontrar el '(', si no lo encontramos significa que la expresión está mal formada.

En caso de que no sea un paréntesis, entonces es un símbolo especial, lo que hacemos es comparar su precedencia con el operador en el tope de la pila, eso lo hacemos con la función `precedence` que simplemente regresa la precedencia del símbolo, si el operador en el tope de la pila tiene mayor o igual precedencia, lo sacamos y lo agregamos al `output`, esto lo repetimos hasta que encontramos un operador con menor precedencia o hasta que la pila esté vacía, luego agregamos el operador actual a la pila. Al final del recorrido de la cadena, vaciamos la pila al `output`.

5. Parseo completo

La función `parse_regex(const char *str)` coordina todo el proceso y llena la estructura regex:

```

regex parse_regex(const char *str) {
    char *explicit = insert_concatenation(str);
    char *postfix = infix_to_postfix(explicit);

    int len = strlen(postfix);
    regex r;
    r.items = malloc(len * sizeof(token));
    r.size = len;
    r.capacity = len;

    for (int i = 0; i < len; i++) {
        r.items[i].value = postfix[i];
        r.items[i].type = get_token_type(postfix[i]);
    }
    free(explicit);
    free(postfix);
    return r;
}

```

Primero recibimos la expresión original, luego le agregamos las concatenaciones explícitas, luego convertimos esa expresión a postfija, después de eso ya tenemos la expresión lista para transformarla a tokens, lo que hacemos es crear una nueva estructura **regex** con el tamaño de la cadena postfija, luego recorremos la cadena y llenamos el arreglo de tokens con los caracteres de la cadena postfija y su tipo correspondiente usando la función **get_token_type**, al final liberamos las cadenas auxiliares y regresamos la estructura **regex** ya llena.

Algoritmos implementados en nfa.c

1. Algoritmo de Thompson

La función **regex_to_nfa(regex r)** construye el NFA usando el algoritmo de Thompson con una pila de fragmentos, y la regex ya debe de estar en postfija.

El algoritmo procesa la expresión postfija símbolo por símbolo, si el símbolo es un operando, se genera un fragmento básicol, si es un operador, se extraen uno o dos fragmentos de la pila (según el tipo de operador) y se construye un nuevo fragmento:

Carácter normal:

```

static fragment create_char_fragment(char c) {
    state *start = create_state();
    state *accept = create_state();
    add_transition(start, accept, c);
    return (fragment){start, accept};
}

```

En este caso, lo que hacemos es crear un automata básico con un nuevo estado de inicio y un nuevo estado de aceptación, luego agregamos una transición entre esos dos estados con el símbolo del carácter, finalmente regresamos un fragmento con el estado de inicio y el estado de aceptación, lo agregamos a la pila de fragmentos para que pueda ser utilizado por los operadores posteriores.

Concatenación (.):

```

static fragment concatenate_fragments(fragment f1, fragment f2) {
    add_epsilon_transition(f1.accept, f2.start);
    return (fragment){f1.start, f2.accept};
}

```

En este caso, lo que hacemos es agregar una transición epsilon desde el estado de aceptación del primer fragmento al estado de inicio del segundo fragmento, luego regresamos un nuevo fragmento con el estado de inicio del primer fragmento y el estado de aceptación del segundo fragmento.

Unión (—):

```

static fragment alternate_fragments(fragment f1, fragment f2) {
    state *start = create_state();
    state *accept = create_state();
    add_epsilon_transition(start, f1.start);
    add_epsilon_transition(start, f2.start);
    add_epsilon_transition(f1.accept, accept);
    add_epsilon_transition(f2.accept, accept);
    return (fragment){start, accept};
}

```

En este caso, lo que hacemos es crear un automata básico con un nuevo estado de inicio y un nuevo estado de aceptación, luego agregamos transiciones epsilon desde el nuevo estado de inicio a los estados de inicio de ambos fragmentos, y también agregamos transiciones epsilon desde los estados de aceptación de ambos fragmentos al nuevo estado de aceptación, finalmente regresamos un nuevo fragmento con el nuevo estado de inicio y el nuevo estado de aceptación.

Kleene (*):

```

static fragment kleene_fragment(fragment f) {
    state *start = create_state();
    state *accept = create_state();
    add_epsilon_transition(start, f.start);
    add_epsilon_transition(start, accept);
    add_epsilon_transition(f.accept, f.start);
    add_epsilon_transition(f.accept, accept);
    return (fragment){start, accept};
}

```

En este caso, lo que hacemos es crear un nuevo estado de inicio y un nuevo estado de aceptación, luego agregamos una transición epsilon desde el nuevo estado de inicio al estado de inicio del fragmento, también agregamos una transición epsilon desde el nuevo estado de inicio al nuevo estado de aceptación para permitir la repetición cero veces, luego agregamos una transición epsilon desde el estado de aceptación del fragmento al estado de inicio del fragmento para permitir la repetición, y finalmente agregamos una transición epsilon desde el estado de aceptación del fragmento al nuevo estado de aceptación para permitir la finalización después de una o más repeticiones, finalmente regresamos un nuevo fragmento con el nuevo estado de inicio y el nuevo estado de aceptación.

Más (+):

```

static fragment plus_fragment(fragment f) {
    state *start = create_state();
    state *accept = create_state();
    add_epsilon_transition(start, f.start);
    add_epsilon_transition(f.accept, f.start);
    add_epsilon_transition(f.accept, accept);
    return (fragment){start, accept};
}

```

En este caso, lo que hacemos es crear un nuevo estado de inicio y un nuevo estado de aceptación, luego agregamos una transición epsilon desde el nuevo estado de inicio al estado de inicio del fragmento para permitir la primera repetición, luego agregamos una transición epsilon desde el estado de aceptación del fragmento al estado de inicio del fragmento para permitir las repeticiones adicionales, y finalmente agregamos una transición epsilon desde el estado de aceptación del fragmento al nuevo estado de aceptación para permitir la finalización después de una o más repeticiones, finalmente regresamos un nuevo fragmento con el nuevo estado de inicio y el nuevo estado de aceptación.

Question (?):

```

static fragment question_fragment(fragment f) {
    state *start = create_state();
    state *accept = create_state();
    add_epsilon_transition(start, f.start);

```

```

        add_epsilon_transition(start, accept);
        add_epsilon_transition(f.accept, accept);
    return (fragment){start, accept};
}

```

En este caso, lo que hacemos es crear un nuevo estado de inicio y un nuevo estado de aceptación, luego agregamos una transición epsilon desde el nuevo estado de inicio al estado de inicio del fragmento para permitir la inclusión del fragmento, también agregamos una transición epsilon desde el nuevo estado de inicio al nuevo estado de aceptación para permitir la exclusión del fragmento, y finalmente agregamos una transición epsilon desde el estado de aceptación del fragmento al nuevo estado de aceptación para permitir la finalización después de incluir el fragmento, finalmente regresamos un nuevo fragmento con el nuevo estado de inicio y el nuevo estado de aceptación.

2. Simulación de NFA

La función `match_nfa(nfa n, const char *str, int len)` simula el NFA usando conjuntos de estados y cerraduras epsilon:

```

int match_nfa(nfa n, const char *str, int len) {
    state_set *current = create_state_set(64);
    epsilon_closure(current, n.start);

    for (int i = 0; i < len; i++) {
        state_set *next = move(current, str[i]);
        free_state_set(current);
        current = next;
        if (current->size == 0) break;
    }

    int accepted = 0;
    for (int i = 0; i < current->size; i++) {
        if (current->states[i] == n.accept) {
            accepted = 1;
            break;
        }
    }
    free_state_set(current);
    return accepted;
}

```

En esta función, primero creamos un conjunto de estados para el estado actual y calculamos la cerradura epsilon del estado inicial del NFA para obtener el conjunto de estados alcanzables sin consumir ningún símbolo. Luego, para cada símbolo en la cadena de entrada, calculamos el conjunto de estados alcanzables desde el conjunto actual a través de transiciones que coinciden con el símbolo, y luego calculamos la cerradura epsilon de ese nuevo conjunto (esto lo hacemos con la función `move`) para obtener el nuevo conjunto de estados actuales. Si en algún punto el conjunto de estados se vuelve vacío, podemos detener la simulación. Al final, verificamos si el estado de aceptación del NFA está en el conjunto de estados actuales para determinar si la cadena es aceptada o no.

La función `epsilon_closure` calcula recursivamente todos los estados alcanzables por transiciones epsilon:

```

static void epsilon_closure(state_set *set, state *s) {
    add_state(set, s);
    for (transition *t = s->transitions; t != NULL; t = t->next) {
        if (t->is_epsilon && !contains_state(set, t->to)) {
            epsilon_closure(set, t->to);
        }
    }
}

```

En esta función, agregamos el estado actual al conjunto de estados, luego iteramos a través de las transiciones del estado. Si encontramos una transición epsilon que lleva a un estado que aún no está en el conjunto, llamamos recursivamente a `epsilon_closure` para ese estado destino, lo que nos permite explorar todos los estados alcanzables a través de transiciones epsilon.

Flujo general

Finalmente flujo completo del programa es:

Input → Concatenación → Postfijo → Tokens → NFA → Match

Ejemplo con "ab|c":

1. `insert_concatenation("ab|c")` → "a.b|c"
2. `infix_to_postfix("a.b|c")` → "ab.c|"
3. `parse_regex` crea tokens: [CHAR:a, CHAR:b, DOT, CHAR:c, PIPE]
4. `regex_to_nfa` construye el NFA con el algoritmo de Thompson
5. `match_nfa` verifica si un string es aceptado

Dificultades encontradas.

Dificultad 1: Salto de línea en las pruebas

Cuando estaba probando el programa con el mini "debugger", ví que la función `fgets` incluía el carácter de salto de línea `\n` al final del input. Esto causaba que al probar `insert_concatenation("abb")`, el resultado fuera "a.b.b.`\n`" en lugar de "a.b.b".

Simplemente agregué una línea para eliminar el salto de línea después de leer la entrada:

```
input[strcspn(input, "\n")] = '\0';
```

Dificultad 2: Concatenación con espacios

En las pruebas de `needs_concatenation`, observamos que cuando había espacios en la entrada, se agregaban concatenaciones incorrectas. Por ejemplo, "aab a" se convertía en "a.a.b. .a", es decir, tomaba el espacio en blanco como un carácter más.

Simplemente agregué una verificación:

```
if (c1 == ' ' || c2 == ' ') return 0;
```

Dificultad 3: Orden de declaración de funciones

Al compilar, aparecían errores de funciones no declaradas porque las funciones auxiliares estaban definidas después de ser utilizadas, principalmente me pasó con las funciones del stack, las declaré después y al probar no me funcionaba nada.

Simplemente agregamos declaraciones forward al inicio del archivo:

```
char* insert_concatenation(const char *str);
char* infix_to_postfix(const char *str);
```

Dificultad 4: Manejo de operadores en Shunting Yard

También al encontrar un paréntesis de cierre `)`, se hacía `symbols.top--` incondicionalmente, lo que podía causar que el índice quedara en -2 si no había `(` en la pila.

Simplemente agregué una verificación antes de descartar el paréntesis.

Dificultad 5: Epsilon-closure recursivo

La función `epsilon_closure` necesita evitar ciclos infinitos cuando hay ciclos de transiciones epsilon en el NFA.

Simplemente implementé una estructura `state_set` que mantiene un conjunto de estados y verifica si un estado ya fue visitado antes de agregarlo:

```
static int contains_state(state_set *set, state *s) {
    for (int i = 0; i < set->size; i++) {
        if (set->states[i] == s) return 1;
    }
    return 0;
}
```

Dificultad 6: Validador en Mac ARM

El validador de la práctica es un binario x86-64, lo que causaba problemas de compatibilidad en Mac, aún con docker tuve que agregar una pequeña flag para poder compilarlo.

```
docker build --platform linux/amd64 -t regex_to_nfa_validator .
docker run --platform linux/amd64 --rm regex_to_nfa_validator
```

Pruebas y resultados.

Sistema de pruebas interactivo

Implementé un pequeño "debugger" interactivo para probar cada función individualmente. El programa se compila con la macro `TEST` definida, lo que activa un menú de pruebas en la función `main`:

```
gcc -DTEST regex.c -o test && ./test
```

Opciones disponibles:

- 1 <char> - Prueba `is_normal_char`
- 2 <char1><char2> - Prueba `needs_concatenation`
- 3 <regex> - Prueba `insert_concatenation`
- 4 <regex> - Prueba `infix_to_postfix`
- q - Salir

Pruebas del programa principal

En mi caso tuve que ajustar el comando de docker porque no funcionaba el validador en mi MAC, pero después de eso pude correrlo sin problemas y todas las pruebas pasaron correctamente. El comando que utilicé fue:

```
$ docker build --platform linux/amd64 -t regex_to_nfa_validator .
$ docker run --platform linux/amd64 --rm regex_to_nfa_validator
```

Resultados del validador:

```
[admin@MacBook-Air-de-admin] - [~/Documents/personal/Escuela/Compiladores/practicas/regex_to_nfa] - [dom. mar 01, 14:52]
└─[$] <git:(development*)> docker build --platform linux/amd64 -t regex_to_nfa_validator
[+] Building 3.1s (9/9) FINISHED
  => [internal] load build definition from Dockerfile
  => transferring dockerfile: 417B
  => [internal] load metadata for docker.io/library/alpine:latest
  => [internal] load .dockerignore
  => transferring context: 2B
  => [1/4] FROM docker.io/library/alpine:latest@sha256:25109184c71bdad752c8312a8623239686a9a2071e8825f20acb8f2198c3f659
  => => resolve docker.io/library/alpine:latest@sha256:25109184c71bdad752c8312a8623239686a9a2071e8825f20acb8f2198c3f659
  => [internal] load build context
  => => transferring context: 21.53kB
  => CACHED [2/4] RUN apk add --no-cache gcc musl-dev make cmake
  => CACHED [3/4] WORKDIR /app
  => CACHED [4/4] COPY . .
  => exporting to image
  => => exporting layers
  => => exporting manifest sha256:ee1a3566e6f703f12033fd24e3c3d7d3ec5c98366bf737140b826781bf6a457
  => => exporting config sha256:6507a0267ebbbf5db4578cefca9a5f93b3c96efce0c9166e13882678fcc3b2449
  => => exporting attestation manifest sha256:a4b317c513c1c4b857073589a0dd5d5bd871d04b151308a424ed25afc565481
  => => exporting manifest list sha256:51775fd1036aff26d3ae91988452bb7c6210db4275f8ead55c9ac63100b99d6
  => => naming to docker.io/library/regex_to_nfa_validator:latest
> [admin@MacBook-Air-de-admin] - [~/Documents/personal/Escuela/Compiladores/practicas/regex_to_nfa] - [dom. mar 01, 16:31]
└─[$] <git:(development*)> docker run --platform linux/amd64 --rm regex_to_nfa_validator
-- The C compiler identification is GNU 15.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Configuring done (1.0s)
-- Generating done (0.0s)
-- Build files have been written to: /app
[ 25%] Building C object CMakeFiles/regex_to_nfa.dir/src/main.c.o
[ 50%] Building C object CMakeFiles/regex_to_nfa.dir/src/nfa.c.o
[ 75%] Building C object CMakeFiles/regex_to_nfa.dir/src/regex.c.o
[100%] Linking C executable regex_to_nfa
[100%] Built target regex_to_nfa
Regex: alba*b(alb)*
  postfix ok
  nfa ok: 11100
Regex: a(blc)*d
  postfix ok
  nfa ok: 1110
Regex: (alb)*c
  postfix ok
  nfa ok: 110
Regex: ab
  postfix ok
  nfa ok: 100
Regex: albic
  postfix ok
  nfa ok: 1110
Regex: a*
  postfix ok
  nfa ok: 1110
Regex: ab+c
  postfix ok
  nfa ok: 1110
Regex: a?b
  postfix ok
  nfa ok: 110
Regex: (ab)*
  postfix ok
  nfa ok: 1110
Regex: (alb)c+d
  postfix ok
  nfa ok: 1110

Overall: PASS
> [admin@MacBook-Air-de-admin] - [~/Documents/personal/Escuela/Compiladores/practicas/regex_to_nfa] - [dom. mar
```