UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE CIENCIAS

Proyecto 01: Construcción de un Analizador Léxico

Compiladores

Integrantes:

Maryam Michelle Del Monte Ortega 320083527
Sahara Mariel Monroy Romero 320206391

Profesor: Manuel Soto Romero
Ayudantes: Diego Méndez Medina
Fausto David Hérnandez Jasso
José Alejandro Pérez Márquez
Jose Manuel Evangelista Tiburcio

Introducción y motivación

El objetivo de este reporte es documentar nuestra construcción de un analizador léxico en Haskell mediante el pipeline: ER $\rightarrow$ AFN $-\varepsilon \rightarrow$ AFN $\rightarrow$ AFD $\rightarrow$ AFDmin $\rightarrow$ MDD $\rightarrow$ lexer,

con política de maximal munch y retroceso al último estado final. Se justifica cada decisión de diseño, se describen las estructuras de datos y se explican los algoritmos de transformación, así como otros detalles de implementación.

1. Diseño de la representación de expresiones regulares

Se define el tipo algebraico Expresion con cuatro constructores básicos: símbolo atómico o caracter (`Term`), concatenación (`Concat`), disyunción (`Or`) y estrella de Kleene (`Star`). Esto nos permite directamente desglosar el AST, y el instance Show garantiza visualización de las expresiones.

En esta parte, una decisión relevante fue no incluir explícitamente el símbolo ε como constructor en `Expresion`. Esto pensando en dos razones principales:

- Simplificación del árbol sintáctico: al eliminar el caso especial ε, se reduce el número de patrones y casos que deben manejarse tanto en la construcción de autómatas como en transformaciones internas. En particular, se evita tratar derivaciones 'triviales' como $\varepsilon \cdot r = r$ y $r \cdot \varepsilon = r$, así como la ambigüedad del operador estrella sobre ε.
- Delegación de ε a la fase de construcción de autómatas: la presencia de ε-transiciones es más natural y controlable en el AFN-ε. Esto permite mantener el tipo `Expresion` mas sencillo y trasladar la complejidad a la parte de autómatas, que ya provee mecanismos para manejar ε.

2. Definición y operaciones sobre AFN-ε

En esta sección se describen las operaciones básicas necesarias para el uso de el AFN-ε. Esta etapa es fundamental en el pipeline, pues sirve de puente entre las expresiones regulares y autómatas sin ε.

2.1. Estructura de datos y notación

Un AFN-ε se modela por la tupla $A = (Q, \Sigma, \Delta, q_0, F)$, donde:

- Q es el conjunto de estados, representado por enteros.
- Σ es el alfabeto de entrada (símbolos de tipo `Char`).
- $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ es el conjunto de transiciones. En la implementación, cada transición se representa como una terna (p, m, q) donde m es de tipo `Maybe Simbolo`: $m =$ `Just` c para transiciones etiquetadas con $c \in \Sigma$ y $m =$ `Nothing` para transiciones ε.
- $q_0 \in Q$ es el estado inicial.
- $F \subseteq Q$ es el conjunto de estados finales.

2.2. Autómata unidad (`afneUnidad`)

La función `afneUnidad` construye el autómata más pequeño que reconoce exactamente un símbolo. Intuitivamente:

- Tiene dos estados: uno de inicio s y uno de aceptación $f = s + 1$.
- Hay una sola transición $s \xrightarrow{c} f$ etiquetada con el símbolo c.
- Si el autómata lee c, pasa al estado de aceptación. Si lee cualquier otro símbolo, no hay transición y no acepta.

Este parte elemental es la base con la que se arma todo lo demás en la construcción del algoritmo de Thompson. Cada término c de una ER se convierte en un `afneUnidad` y luego se combina con concatenación, disyunción y estrella para formar autómatas más complejos.

2.3. Cerradura ε

La cerradura ε de un conjunto de estados es, en pocas palabras, todo lo que puedes alcanzar moviéndote solo por transiciones ε, sin consumir ningún símbolo, empezando desde ese conjunto. Siempre incluye los estados de partida y todos los que se alcancen encadenando transiciones ε las veces que haga falta, hasta que ya no aparezcan estados nuevos.

En la función `cerraduraEpsilon` lo implementamos de forma directa:

- Mantenemos dos conjuntos: uno de *visitados* y una *frontera* (o sea, estos estados pendientes de procesar).
- Mientras la frontera no esté vacía, tomamos un estado, miramos a dónde podemos ir con ε, y añadimos a la frontera los que aún no estén visitados.
- Al terminar, el conjunto de visitados es la cerradura ε.

2.4. Movimiento con etiquetas

La operación `mover`(A, R, c) regresa los estados a los que se puede llegar desde cualquier estado de R usando una sola transición etiquetada con el símbolo c, sin usar transiciones ε adicionales. Para hacerlo eficiente, construimos una tabla que dado un par (estado, símbolo) nos devuelve los posibles destinos, luego unimos las respuestas para todos los estados de R.

Esta funcion se usa especialmente en:

- Eliminación de ε, combinándola con la cerradura ε antes y después de consumir c.
- En el método de subconjuntos, donde se aplica sobre conjuntos de estados para construir las transiciones del AFD.

2.5. Normalización de estados

La función `normalizarEstados` renombra los estados para que queden como $0, 1, 2, \ldots$ en orden, y además ordena y deduplica alfabeto y transiciones. Esto tiene varias ventajas: facilita comparar resultados en pruebas, evita que la numeración dependa del orden de construcción y prepara el terreno para etapas como minimización y visualización.

2.6. Por qué usamos enteros para los estados y no Strings

Optamos por representar los estados con enteros en lugar de cadenas como q0, q1, etc., por motivos prácticos:

- Comparar, ordenar y usar enteros como llaves en estructuras como `Map` o `Set` es más rápido y sencillo que hacerlo con cadenas.
- Por simplicidad, ya que q0 y 0 expresan lo mismo. Trabajar con números reduce errores y evita lógica extra para generar o interpretar nombres.
- En etapas posteriores (pasar a determinista, minimización, tablas del lexer) es mas facil trabajarse con enteros.

2.7. Otros detalles

- La representación de ε mediante `Nothing` en `Maybe Simbolo` evita introducir un símbolo especial en Σ y mantiene el tipo seguro.

3. Construcción de un AFN-ε a partir de una expresión regular

Aquí describimos el algoritmo que transforma una expresión regular en un AFN-ε usando las construcciones de Thompson que hemos visto en clase. La implementación se organiza en funciones que corresponden directamente a los operadores de la expresión regular: término (símbolo), unión ($|$), concatenación, y estrella de Kleene ($*$).

3.1. Procedimiento

Tenemos que `regexToAfne` es la función principal. Dada una expresión regular e, se hace de manera recursiva:

- Para un término `Term` c, se construye un AFN-ε elemental con dos estados y una única transición etiquetada con c.
- Para `Or` de e_1 y e_2, se construyen los autómatas de e_1 y e_2, y se conectan mediante un nuevo estado inicial y uno final con transiciones ε que conectan hacia e_1 o e_2, y desde los finales de cada uno hacia el nuevo final.
- Para `Concat` de e_1 con e_2, se construyen los autómatas de e_1 y e_2, y se conectan con transiciones ε desde cada estado final de e_1 al estado inicial de e_2, entonces el inicial del resultado es el de e_1 y los finales son los de e_2, tal como vimos en clase.
- Para la estrella `Star` e, se construye el autómata de e y se rodea con un nuevo inicial y un nuevo final; se añaden transiciones ε que permiten:
  - aceptar la cadena vacía (del nuevo inicial al nuevo final),
  - entrar a una iteración (del nuevo inicial al inicial del autómata de e),
  - iterar nuevamente (de cada final de e al inicial de e),
  - salir aceptando (de cada final de e al nuevo final).

Finalmente, se aplica una normalización de estados para dejar los numeros ordenados.

3.2. Gestión de estados recién creados

Para las operaciones que introducen nuevos estados (o sea, unión y estrella) se utiliza un contador `sig` que siempre apunta al próximo identificador libre. Cuando se crea un nuevo par de estados (s, f) (nuevo inicial y nuevo final), se consumen dos valores consecutivos del contador y éste se actualiza en consecuencia. Esto nos ayuda a tener unicos identificadores y evitar pronlemas si queremos unir dos automatas.

3.3. Construcción de cada operador

Término Dado un símbolo c, se construye un AFN-ε con $Q = \{s, f\}, \Sigma = \{c\}, \Delta = \{(s, c, f)\}, q_0 = s, F = \{f\}$, donde s y f son estados recién creados consecutivos. Despues de la construcción, el contador avanza en 2.

Unión Dados dos AFN-ε correspondientes a e_1 y e_2, se crean dos estados nuevos s (inicial) y f (final). Se añaden transiciones ε desde s a cada inicial de los autómatas de e_1 y e_2, y desde cada estado final de ambos hacia f. El conjunto de estados y transiciones es la unión de partes no 'solapadas' de los componentes de ambos autómatas, más los nuevos extremos s, f y las ε-transiciones que conectan. El alfabeto resultante es la unión de alfabetos, eliminando duplicados. El contador avanza en 2.

Concatenación Dados los AFN-ε de e_1 y e_2, se agregan transiciones ε desde cada estado final del primero hacia el estado inicial del segundo. El estado inicial del resultado es el inicial del primero, y el conjunto de estados finales es el de los finales del segundo. El alfabeto es la unión de alfabetos con eliminación de duplicados. En esta operación no se consumen estados nuevos recien creados adicionales.

Estrella de Kleene Dado el AFN-ε de e, se introducen dos estados nuevos s y f. Se añaden:

$s \xrightarrow{\varepsilon} f, s \xrightarrow{\varepsilon} q_0, x \xrightarrow{\varepsilon} q_0 \quad x \xrightarrow{\varepsilon} f$ para todo $x \in F$, donde q_0 y F son el inicial y los finales del autómata de e. Con esto ya aceptamos la cadena vacía, una o más repeticiones y podemos terminar tras cualquier repetición. El contador avanza en 2.

3.4. Combinar alfabetos y transiciones

Cuando se combinan subautómatas, el alfabeto resultante es la unión de los alfabetos de cada componente, eliminando símbolos repetidos. Las transiciones del resultado son la concatenación de las transiciones de los

operandos más las nuevas ε-transiciones que conectan la estructura (por ejemplo, de finales a iniciales en concatenación, o desde y hacia los nuevos extremos en unión y estrella).

3.5. Normalización

El autómata resultante lo normalizamos donde volvemos a enumerar estados para ordenar y eliminar duplicados en el alfabeto y de las transiciones

4. Definición de AFN

4.1. Estructura del AFN

Un AFN se modela por: Q (estados), Σ (alfabeto), Δ (transiciones), q_0 (inicial), F (finales)

Las transiciones tienen forma (p, c, q) con $p, q \in Q$ y $c \in \Sigma$. El tipo `AFN` agrupa estos componentes y permite compararlos e imprimirlos con `Eq` y `Show`.

4.2. Operación `mover`

Dado un AFN A, un conjunto de estados R y un símbolo c, queremos saber a qué estados podemos ir en un solo paso usando c.

Definición clara: $\texttt{mover}(A, R, c) = \{\, q$ tal que existe $p \in R$ con una transición $(p, c, q) \in \Delta\}$. Cómo se implementa:

- Primero se crea una tabla que para cada par (p, c), guarda la lista de destinos $[q]$.
- Luego, por cada estado p en R, se busca la entrada (p, c) y se juntan todos los q encontrados.
- El resultado se devuelve como un conjunto, para no repetir estados.

5. Eliminación de ε y construcción del AFN equivalente

Queremos convertir un AFN con transiciones ε en otro AFN sin ellas, pero que acepte las mismas cadenas. La idea clave es 'absorber' los efectos de ε en dos lugares: al moverse por símbolos y al decidir qué estados son finales.

5.1. Función `afnEpToAfn`

- Entrada: un AFN-ε.
- Salida: un AFN sin transiciones ε que reconoce el mismo lenguaje.

Para hacerlo eficiente y claro, usamos:

- Conjuntos (Data.Set) para evitar duplicados (igual que en los otros modulos).
- Listas (Data.List) para ordenar y limpiar.
- Mapas (Data.Map.Strict) para consultar rápido destinos y cerraduras.

5.2. Cerradura ε por estado

La cerradura ε de un estado la guardamos en una tabla: `tablaCierre` : $p \mapsto \text{cerradura}_\varepsilon(\{p\})$

Así, cuando necesitemos la cerradura de p, la obtenemos en una consulta. Si falta alguna entrada, usamos por defecto $\{p\}$.

5.3. Movimiento por símbolo, incorporando ε

Para movernos por un símbolo c desde un conjunto de estados R, primero hacemos lo obvio, que es mirar las transiciones etiquetadas con c. Montamos una tabla: $(p, c) \mapsto \{q \mid (p, c, q)$ es una transición no-$\varepsilon\}$

Pero como existen ε, no debemos partir desde p directo, si no desde todo lo que p alcanza por ε. Y además, al llegar, también cerramos por ε. Queda: $\text{destino}(p, c) = \text{cerradura}_\varepsilon\Big(\text{mover}\big(c, \text{cerradura}_\varepsilon(\{p\})\big)\Big)$.

5.4. Estados finales en el AFN resultante

Un estado p será final si, desde p, puedes aceptar sin consumir símbolos adicionales (usando solo ε). Es decir: $p \in F' \quad \text{si} \quad \text{cerradura}_\varepsilon(\{p\}) \cap F \neq \emptyset$.

O sea, si desde p puedes caer por ε en algún final, entonces p ya es final en el nuevo AFN.

5.5. Generar las nuevas transiciones

Para cada par (p, c):

1. Calcular $\text{destino}(p, c)$ con la composición anterior.
2. Por cada $q \in \text{destino}(p, c)$ agregar la transición $p \xrightarrow{c} q$.

Al terminar:

- Ordenar y eliminar duplicados en transiciones, estados y alfabeto.
- Mantener el estado inicial original.

6. Definición AFD

A diferencia del AFN, aquí cada par (estado, símbolo) tiene a lo sumo un destino, por eso la operación mover devuelve un único estado o nada.

6.1. Estructura del AFD

El tipo de datos del AFD agrupa: Q (estados), Σ (alfabeto), Δ (transiciones), q_0 (inicial), F (finales).

- Estados: enteros (type Estado = Int).
- Símbolos: caracteres (type Simbolo = Char).
- Transiciones: tuplas (q, c, q) donde ir desde q con c lleva a q.
- El registro AFD junta listas de estados, alfabeto, transiciones, inicial y finales, y deriva `Eq` y `Show` para comparar e imprimir.

6.2. Operación mover

Dado un AFD A, un estado q y un símbolo c, obtenemos el destino único si existe.

La función mover toma un autómata AFD, un estado actual y un símbolo de entrada. Busca si existe una transición que salga exactamente desde ese estado con ese símbolo. Si la encuentra, devuelve el estado destino de esa transición. Si no existe tal transición, indica que no hay un siguiente estado (devuelve Nothing). En un AFD, para cada par (estado, símbolo) puede haber a lo sumo una transición, así que el resultado, cuando existe, es único.

Entonces se construye una tabla (o mapa) que asocia cada par (q, c) con su destino q:

tabla : $(q, c) \mapsto q$.

Se consulta (q, c); si está, se devuelve q, si no, `Nothing`.

6.3. Normalización de estados

Aquí también normalizamos con el objetivo de ordenar y renumerar los estados para dejar el AFD en una forma estable.

Pasos:

1. Recolectar y ordenar estados: Miramos todos los estados que aparecen en el autómata (en la lista de estados, en las transiciones, en el inicial y en los finales). Quitamos duplicados y los ordenamos. Esto nos da una lista limpia y en orden estable.
2. Asignamos nuevos IDs consecutivos A los estados ordenados les damos nuevos nombres: 0, 1, 2,

   Guardamos esa correspondencia en una tabla de renombrado para poder transformar todo de forma consistente.
3. Ahora sí aplicamos el renombrado de un jalón:
   - Estado inicial: cambiamos su nombre usando la tabla.
   - Estados finales: cambiamos cada uno usando la misma tabla.
   - Transiciones: cada flecha (p, c, q) se convierte en (ren(p), c, ren(q)). Así garantizamos que todo el AFD usa la nueva numeración compacta.
4. Eliminamos símbolos repetidos en el alfabeto.

7. Conversión de AFN a AFD

Este módulo implementa la construcción por subconjuntos que vimos en clase: convierte un autómata finito no determinista (AFN) en un autómata finito determinista (AFD). La idea central es que cada estado del AFD representa un conjunto de estados del AFN. De este modo, al leer un símbolo, el AFD mueve el conjunto actual al conjunto de todos los posibles destinos en el AFN.

- Un estado del AFD equivale a estar en cualquiera de estos estados del AFN. O sea es un conjunto de estados del AFN.
- La transición del AFD con un símbolo c desde ese conjunto se calcula aplicando mover del AFN en paralelo a todos los estados del conjunto, y uniendo los destinos en un nuevo conjunto.

  Si ese conjunto destino ya existe en el AFD, se reutiliza su identificador, pero si no existe, se crea un nuevo estado del AFD.
- Los estados finales del AFD son aquellos conjuntos que contienen al menos un estado final del AFN.

7.1. Cómo se construye en el código

- Se toma el alfabeto del AFN y se elimina duplicado.
- Se define el estado inicial del AFD como el conjunto que sólo contiene el estado inicial del AFN.
- Se realiza un recorrido tipo BFS sobre conjuntos donde se mantiene una cola de conjuntos por procesar.

  Se tiene un map que asigna a cada conjunto un identificador de estado del AFD.

  Para cada conjunto y cada símbolo del alfabeto, se calcula su conjunto destino usando la operación mover del AFN. Si el conjunto destino es nuevo, se registra con un nuevo ID y se agrega a la cola y si ya existe, se usa su ID.

  Se registra la transición desde el ID del conjunto actual con el símbolo, hacia el ID del conjunto destino.
- Al terminar el recorrido los estados del AFD son todos los IDs asignados y las transiciones son las que se acumularon en el map de transiciones.

  El inicial es 0 (el ID del conjunto inicial). Los finales son aquellos IDs cuyo conjunto contiene al menos un estado final del AFN.
- Finalmente, se normaliza el AFD.

8. AFDmin: Minimización del Autómata Finito Determinista

A partir de esta etapa, las definiciones y las implementaciones de un paso a otro se integraron dentro de un mismo módulo. Esto se hizo con el propósito de reducir la cantidad de archivos en el proyecto.

En esta parte del proyecto se implementó la función `minimizar`, que toma un AFD completo y devuelve una versión equivalente con el menor número de estados posible. La idea principal fue simplificar el autómata sin cambiar el lenguaje que reconoce.

La implementación se divide en pasos pequeños y claros:

- **Eliminar inalcanzables:** Se quitan los estados a los que nunca se puede llegar desde el inicial. Esto hace que el autómata sea más limpio y fácil de analizar.
- **Completar con nodoMuerte:** Si falta alguna transición, se agrega un estado nodoMuerte que absorbe todos los casos no definidos. Esto asegura que el AFD quede completamente definido.
- **Refinamiento de particiones:** Se agrupan los estados que se comportan igual (tienen las mismas transiciones hacia los mismos tipos de estados). El algoritmo va refinando estos grupos hasta que no haya más cambios.
- **Reconstrucción:** Cada grupo final se convierte en un nuevo estado, generando el AFD minimizado.

El resultado es un AFD reducido y estable que servirá como base para los siguientes pasos del proyecto (MDD y lexer).

9. MDD: Construcción de la Máquina Discriminadora a partir de los AFDmin

Después de minimizar cada autómata con `AFDMin`, el siguiente paso del proyecto fue construir la **MDD (Máquina Discriminadora Determinista)**. La MDD combina todos los AFDmin en una sola máquina que permite al *lexer* analizar la entrada de y decidir qué token reconocer en cada momento.

Cada AFDmin representa las reglas para un token individual. Sin embargo, el *lexer* necesita procesar todo el código fuente en un único recorrido, no correr varios autómatas por separado. Por eso, la MDD **integra todos los AFDmin** y **discrimina** cuál token corresponde a cada secuencia leída.

Cómo se construye la MDD

La implementación de `buildMDD` sigue lo sigunte:

- **Unificación del alfabeto:** Se crea un alfabeto global (`sigma`) con todos los símbolos que aparecen en los distintos AFDs.
- **Totalización:** Antes de combinar los AFDmin, cada uno se completa con un estado nodoMuerte para que todas las transiciones estén definidas. Esto evita errores y garantiza que la MDD sea completamente determinista.
- **Estados como vectores:** Cada estado de la MDD representa un **vector de estados**, uno por cada AFDmin. Por ejemplo, si hay tres AFDs, un estado podría verse como $[q_1, q_2, q_3]$, donde cada componente es el estado actual de un token distinto.
- **Exploración y generación de transiciones:** A partir del vector inicial (formado con los estados iniciales de todos los AFDs), se exploran todas las posibles transiciones para cada símbolo del alfabeto. Si aparece un nuevo vector, se le asigna un número de estado y se sigue expandiendo hasta cubrir todo el espacio alcanzable.
- **Etiquetado de estados finales:** Un estado de la MDD se marca como final si al menos una de sus componentes pertenece a un estado final de su AFD correspondiente. Si hay varios posibles tokens, se aplica una **prioridad** (por ejemplo, el orden en que se definieron) para decidir cuál token reconocer primero.

10. Lexer: Generación del analizador léxico a partir de la MDD

Una vez construida la MDD, el siguiente paso fue implementar el **lexer**, que usa esta máquina para recorrer el texto de entrada y producir los tokens correspondientes. El lexer ya no necesita ejecutar varios autómatas

por separado, sino que se apoya directamente en la MDD para decidir qué token reconocer en cada punto del código fuente.

Cómo funciona el lexer

La función principal `lexerM` toma una MDD y una cadena de entrada, y devuelve una lista de pares `(token, lexema)`. El proceso sigue la idea de recorrer la entrada carácter por carácter mientras se avanza dentro de la MDD:

- **Recorrido con la tabla de transiciones:** Se genera un mapa con las transiciones de la MDD. Por cada símbolo leído, el lexer se mueve al siguiente estado según la tabla.
- **Reconocimiento de tokens:** Si se llega a un estado final, se guarda el lexema reconocido junto con el token que representa (usando las etiquetas definidas en la MDD).
- **Regla de *maximal munch*:** El lexer continúa avanzando mientras existan transiciones válidas y solo detiene el reconocimiento cuando ya no puede seguir. En ese momento, devuelve el último token válido encontrado y reanuda el proceso desde el resto de la cadena.
- **Manejo de errores:** Si se encuentra un símbolo que no pertenece al alfabeto, se lanza un error léxico indicando el carácter inesperado.

Construcción del lexer desde expresiones regulares

La función `buildLexerFromRegex` hace todo el proceso: toma una lista de pares `(token, expresión regular)` y convierte cada expresión en un AFD minimizado usando las etapas Luego, todos esos AFDmin se combinan con `buildMDD` para formar la MDD global, que finalmente se pasa a `lexerM`.

De esta forma, el lexer puede reconocer todos los tokens definidos en el lenguaje de forma determinista y con un solo recorrido sobre la entrada.

11. Lenguaje IMP: Definición y lectura

El lenguaje IMP se define en el archivo `IMP.md`, donde se describen los tokens del lenguaje mediante expresiones regulares. Cada línea asocia un nombre de token con su expresión.

Entre los tokens principales definidos están:

- **num:** reconoce números como `0`, `15` o `-42`.
- **ident:** identifica nombres de variables o funciones.
- **asign:** el operador de asignación `:=`.
- **opArit:** operadores aritméticos como `+`, `-`, `*`, `/`.
- **opRel:** operadores relacionales `<`, `>`, `=`.
- **opBool:** operadores booleanos como `not`, `and`, `or`.
- **bool:** los valores lógicos `true` y `false`.
- **reservCond, reservCiclo, reservSkip:** las palabras reservadas del lenguaje como `if`, `then`, `else`, `while`, `do`, `for`, `skip`.
- **puntuacion y delim:** los símbolos `;`, `{ }`, `( )`.
- **WS (whitespace):** reconoce espacios, tabulaciones y saltos de línea. Este token no representa un símbolo del lenguaje, pero se incluye para que el analizador léxico pueda avanzar correctamente ignorando los espacios en blanco entre tokens válidos.

Lectura y procesamiento del archivo `IMP.md`

El `main` del proyecto inicia leyendo el archivo `IMP.md` ubicado en la carpeta `specs/`. Primero se eliminan los comentarios con la función `borraComentarios`, que detecta y borra los bloques marcados con `*- -*` o `**`

`**`. Luego, `buildTokensFromSpecs` analiza el texto limpio, extrae los nombres de los tokens (las palabras antes del signo igual `=`) y construye la lista `impTokens`, que asocia cada nombre con su expresión regular correspondiente.

Ejecución del programa principal

El `main` muestra un pequeño menú con ejemplos de programas IMP para probar el analizador. Al seleccionar uno, el programa:

1. Lee el archivo de ejemplo y muestra su contenido original.
2. Quita los comentarios del código fuente.
3. Analiza cada línea para marcar si tiene o no comentarios.
4. Se aplica el lexer sobre el código sin comentarios.
5. Muestra todos los tokens encontrados, incluyendo espacios y símbolos desconocidos.

Y así finalmente simulamos un analizador léxico con nuestro lenguaje IMP :D.