CONCEPTOS CLAVE:

Buffer: “pizzara” temporal. El scanner lo usa para guardar el texto del token mas reciente que se acaba de leer del archivo.

**Proceso del código:**

1. TOKEN -> enumeración de las etiquetas posibles con las que clasificar palabras.
2. Chequeo de errores
3. Objetivo() -> Inicia la compilación

Verifica y ejecuta:

1. El programa / 2. Existencia de FinDeTexto / 3. Terminar() (la compilación)

1. Programa() - >Analiza lo que esta en el cuerpo del código entre INICIO y FIN
2. Comenzar() -> Actualmente vacia. Sirve para configuraciones iniciales si fuera necesario
3. Match(TOKEN t) -> recibe el token que se espera. Math(INICIO) espera ver la palabra “inicio”.
   1. **ProximoToken()** -> devuelve el token que el scanner acaba de leer.
      1. Si el **flagToken es 0 (el token anterior, si hubo, ya fue usado por match)**, **busca el nuevo token** con tokenActual = scanner()
         1. Si el tokenActual es ERRORLEXICO, se llama a la función ErrorLexico() que se encarga de avisar sobre errores.
      2. **Scanner()** -> devuelve un TOKEN.

La tabla es el “**cerebro**”. Las filas son el estado actual y las columnas son el tipo de carácter que acaba de leer.

* Arranca en el estado 0, al igual que el contador i
* Lee un caracter del archivo: car = fgetc(in).
* Categoriza el caracter: col = columna(car).
* En base al estado en el que estaba (inicialmente 0) y la columna (el tipo de carácter leído) se busca en la tabla su posición correspondiente.
* Si el **carácter no es un espacio**, se guarda en el **buffer** y el contador i avanza +1.
* Esto se **repite** hasta que se alcanza un **estado final**
* Según el estado al que llego se toma una acción distinta.
  + 2: lo guardado en el buffer es un ID
  + 4: … es una CONSTANTE
    - En estos se usa ungetc para retroceder un carácter. En el caso del ID y de la CONSTANTE se saca el ; . Ademas en el buffer esta carácter “extra” es reemplazado por un carácter nulo/terminador \0
  + 5: Es una SUMA
  + 6: Es una RESTA
  + 7: es un PARENIZQUIERDO
  + 8: es un PARENDERECHO
  + […]

Siguiendo con **ProximoToken()** . Tras leer este token y comprobar que es correcto. El flagToken pasa a 1, lo que significa que hay un token que aun no fue verificado.

Entonces si el tokenActual es una ID llama a **buscar(buffer, TS, &tokenActual)** para verificar si esta palabra está en la tabla de símbolos como una palabra reservada.

1. **Buscar(char \*id,RegTS \*TS, TOKEN \*t)** -> determina si un identificador esta en la TS. Recibe la palabra a buscar (id), un puntero a la TS y un puntero a una variable TOKEN

**RegTS** -> estructura de datos. Cada fila de la misma se compone de:

* Identifi: char[] que almacena el texto de la palabra
* t: la etiqueta TOKEN asociada a esa palabra
  1. Mediante un contador i y un while se recorre la TS desde el principio. Mientras el texto en esa posición de la TS (identifi) no sea $ ($ **es un centinela/marca de fin de tabla**).

(strcmp da 0 (false) si ambas cadenas son iguales).

Si el id que recibió la función esta en la TS, entonces actualiza t con el tipo de token que estaba guardado en la tabla y retorna TRUE

SINO, sigue avanzando en el bucle. Si no encuentra coincidencia, devuelve false.

Entonces en cualquier caso que no sea un errorlexico, ProximoToken devuelve el tokenActual.

SI el token t que recibió Match no coincide con el próximo tokken, se produce un error sintactico. Si no, el flagToken se establece en 0.

1. **ListaSentencias() ->** analiza el cuerpo principal del programa (lo que esta entre inicio y fin)
   1. Llama a Sentencia() (al menos una vez) -> encargada de analizar 1 sentencia
   2. Luego entra en un bucle infinito. Según cual sea el proximoToken, se toma una decisión mediante un switch:
      1. ID, LEER, ESCRIBIR, se llama a sentencia() nuevamente para el análisis
      2. En cualquier otro caso (default) significa que la lista de sentencias se termino y se debe salir de la función ListaSentencias (vuelve al Programa()
2. **Sentencia()** -> busca “averiguar” cual de los **3 tipos de sentencias** es
   1. Toma el primer token: TOKEN *tok* *=* *ProximoToken*();

**REG\_EXPRESION** -> plantilla que se usa para guardar toda la información importante sobre una “expresión”. Se compone de:

* TOKEN clase -> le dice al compilador que tipo de cosa es
* Char Nombre [] -> guarda el texto de la cosa (si es num, lo guarda como texto, si es una operación, guarda el nombre del temp que la almacena)
* Int valor -> si es una CONSTANTE, guarda su valor numerico real.
  1. Crea REG\_EXPRESION izq, der. Fichas vacias que se rellenaran. Izq guarda la ficha de la variabla a la izquierda del := y der el resultado final
  2. Median un switch se decide como actuar según el tipo de TOKEN
     1. ID:

Estamos ante algo como A := B+5 (<expresión>)

* + - 1. Llama a **Identificador** para izq

**Identificador(REG\_EXPRESION \*presul)** -> procesa el nombre de una variable (ID) que la gramática espera encontrar en ese momento

Verifica con Match(ID) que el token actual sea un ID (si no se produce un errorsintactico). Si esto es exitoso, se llama a la rutina semántica ProcesarId(), y la “ficha” que sale de ese proceso se guarda en el espacio de memoria de presul.

**REG\_EXPRESION** **ProcesarId()** -> rutina semántica.

Crea una ficha vacia: REG\_EXPRESION reg

Verifica la TS con Chequear(Buffer) para saber si la palabra en el buffer ya esta en dicha TS.

* **Chequear(char \*s) ->** si la cadena NO esta en la TS, la agrega con la funcion **Colocar(s, TS)** y muestra un mensaje en pantalla
* **Colocar(char \*id,RegTS \*TS)** -> agrega un identificador en la TS. Inicializa un contador i=4 (las posiciones de 0 a 3 estan reservadas por inicio, fin, leer y escribir), y corre un while que se ejecuta mientras en la posición i de la TS.identifi no se encuentre el centinela “$”. Mientras eso no se cumpla i +1

Una vez terminado el while, se comprueba si i<999. Si se cumple entonces se compia el id en la TS[i].identifi y se le asigna al TS[i].t el token ID. Finalmente se mueve al siguiente espacio y pone el centinela “$” alli.

Una vez realizado el chequeo, y la colocación si fue necesario, en la clase del reg se le asigna el token ID, se copia en reg.nombre el nombre de la variable (**RELLENAR LA FICHA**). Finalmente retorna el registro.

* + - 1. Volviendo a sentencia() luego de pasar por Identificador(). Se hace un Match(Asignacion), con el que se verifica el :=
      2. Se analiza la expresión, invocando a Expresion(&der)

**Expresion(REG\_EXPRESION \*presul) ->** analiza expresiones

* Crea dos registros de expresión, uno para cada operando (Izq y Der)
* Op guardara el operado y token T el token
* Se ejecuta Primaria(&OperandoIzq) *(funcionamiento explicado más abajo, resumidamente obtiene la primera parte de la expresión, por ejemplo, en A+B-C, obtiene A).*
* Se inicia un bucle for que procesara las partes que siguen de la expresion:
  + Se asigna a t el proximoToken.
  + Mientras t sea una SUMA o RESTA el bucle avanza
  + Al finalizar cada iteración, se le asigna a t el proximoToken

En cada iteración del bucle, se invoca opedadorAditivo enviándole el op:

***OperadorAditivo*(*char* *\**presul) ->** verifica que efectivamente el token actual es SUMA o RESTA

*Recibe el proximoToken y lo guarda en t, si t es uno de los susodichos tokens, se hace un match(t) para procesarlo y se copia en presul el contenido del buffer mediante la función* ***procesarOp().***  *Si no es el caso, da un errorSintactico*

Ahora op tiene el símbolo correspondiente.

Se invoca a Primaria(&operandoDer)

**Primaria(REG\_EXPRESION \*resul)** -> es la “calculadora”

Guarda en tok el proximoToken y en base a el decide como actuar por medio de un switch.

Si es un ID, invoca a Identificador para procesarlo y escribir en presul el resultado

Si es una CONSTANTE, la procesa con Match(CONSTANTE) y asigna a presul el resultado de ProcesarCte()

**REG\_EXPRESION ProcesarCte(void)**

Crea un registro (reg) de tipo REG\_EXPRESION. A su clase se le asigna el token CONSTANTE y se copia en nombre el texto presente en el buffer. Si el contenido en el buffer es numerico, se guarda su valor en el campo valor del registro. La función retorna el registro completo.

Si es un PARENIZQUIERDO lo consume con match(PARENIZQUIERDO) y **VUELVE** a invocar a Expresion con **Expresion(presul)** con el objetivo de analizar la expresión interna que esta entre los paréntesis. Se repite el proceso que venimos describiendo en estos últimos párrafos. Una vez analizada la expresión se verifica y consume el paréntesis derecho con Match(PARENDERECHO)

Default:si no es nada de lo anterior. Estamos ante un error sintactico.

* Luego se le asigna a operandoIzq el registro que retorna la función GenInfijo(operandoIzq, op, operandoDer);

**REG\_EXPRESION GenInfijo(REG\_EXPRESION e1, char \*op, REG\_EXPRESION e2) ->**  genera la instrucción para una operación infija y construye un registro semántico con su resultado (variable temporal con la que guardar el resultado).

Se llama siempre que se encuentra un + o -

Crea un registro(reg) nuevo y vacio.

numTemp es una variable que **conserva su valor entre llamadas a la función** (la primera vez vale1, la segunda 2, etc). cadTemp define la base del nombre para las variables temporales, y cadNum y cadOp son variables auxiliares de texto.

Se traduce el operador. Si es – en cadOp se pone “Restar” y si es + “Sumar. El numTemp es pasado a texto y agregado en cadNum. Entonces el numTemp+1 para el próximo llamado. Se concatena cadTemp con cadNum ("Temp&1").

Lo que prosigue son dos if de “comprobación de seguridad”, se extrae el nombre (texto) de los registros que recibió la función al ser invocadas con extraer(), y esto se lo pasa a chequear para hacer la comprobación de su existencia en la TS.

Una vez hecha la comprobación, se invoca Chequear(cadTemp): buscara “Temp&1” (o el numero que corresponda) en la TS (Buscar()), y como no lo encontrara, lo va a agregar (Colocar()).

Finalmente, con Generar(*cadOp*, *Extraer*(*&*e1), *Extraer*(*&*e2), *cadTemp) se imprimirá la instrucción indicando la creación de la variable temporal*

***Generar(char \*co, char \*a, char \*b, char \*c) ->*** *produce la salida de la instrucción para la MV por stdout mediante un print f.*

Finalmente se copia en el campo nombre del reg el nombre de la variable temporal y devuelve esta nueva ficha a Expresion.

* + - 1. A presul se le asigna lo obtenido en operandoIzq de GenInfijo,finalizando con la función Expresion.

Volviendo a sentencia, seguimos en el caso de ID. Ahora se invoca a **Asignar(izq, der)** -> que genera la instrucción para la asignación con Generar(“Almacena”, ….).

Finalmente se hace Match(PUNTOYCOMA) para comprobar el final de la sentencia.

* + 1. LEER:
       1. Hace Match(LEER) para consumir el token
       2. Match(PARENIZQUIERDO) para verificar el (
       3. ListaIdentificadores()

**ListaIdentificadores() ->**  procesa la lista de variables dentro de un comando leer(). Significa que, espera un identificador, y mientra siga viendo comas (,) espera otro identificador.

Se inicializa un token t y un registro (reg).

Se procesa la primer variable con identificador(&reg) (el proceso de la función fue descripto mas arriba).

Luego llama a leer pasandole el reg.

**Leer(REG\_EXPRESION in) ->** genera la instrucción para leer

Llama a Generar pasandole (“Read, in.nombre, “Entera”, “”)

Se inicia un bucle for, en el que se el asigna a t el proximoToken, este bucle funcionara mientras t sea una coma (, token COMA) y cada vez que termine una iteración, avanzara hacia el nuevo proximoToken. Por cada iteración se hará Match(COMA), se procesa la siguiente variable con Identificador(&reg) y se genera el código para esa variable con leer(reg).

* + - 1. Match(PARENDERECHO) para verificar el )
      2. Match(PUNTOYCOMA)
    1. ESCRIBIR:
       1. Hace Match(ESCRIBIR) para consumir el token
       2. Match(PARENIZQUIERDO) para verificar el (
       3. ListaExpresiones()

**ListaExpresiones() ->** procesa la lista de cosas dentro de un comando escribir(). IGUAL que ListaIdentificadores pero reemplazando Identificador por Expresion() y Leer() por Escribir()

Llama a Expresion(&reg) para procesar la primera expresión, genera el código con Escribir(reg) y luego con un while, mientras encuentre comas, se avanza en los tokens con ProximoTokens, consumiendo las comas con Math(COMA) y ejecutando Expresion(&reg) y Escribir(Reg)

* + - 1. Match(PARENDERECHO) para verificar el )
      2. Match(PUNTOYCOMA)
    1. Default: no es un token que pueden iniciar una sentencia valida. Se deja de ejecutar sentencia y devuelve el control a la función original (normalmente ListaSentencias()).

1. Una vez terminado de analizar toda la lista de sentencias en ListaSentencias() se vuelve a Programa() y se realiza la ultima validación del programa: Match(FIN).
2. El control vuelve a Objetivo(). Se hace un Match(FDT) para verificar que ya no hay texto en el archivo y se invoca a Terminar()
3. **Terminar()** -> Se genera la instrucción de finalización, con Generar(“Detiene”,””,””,””)