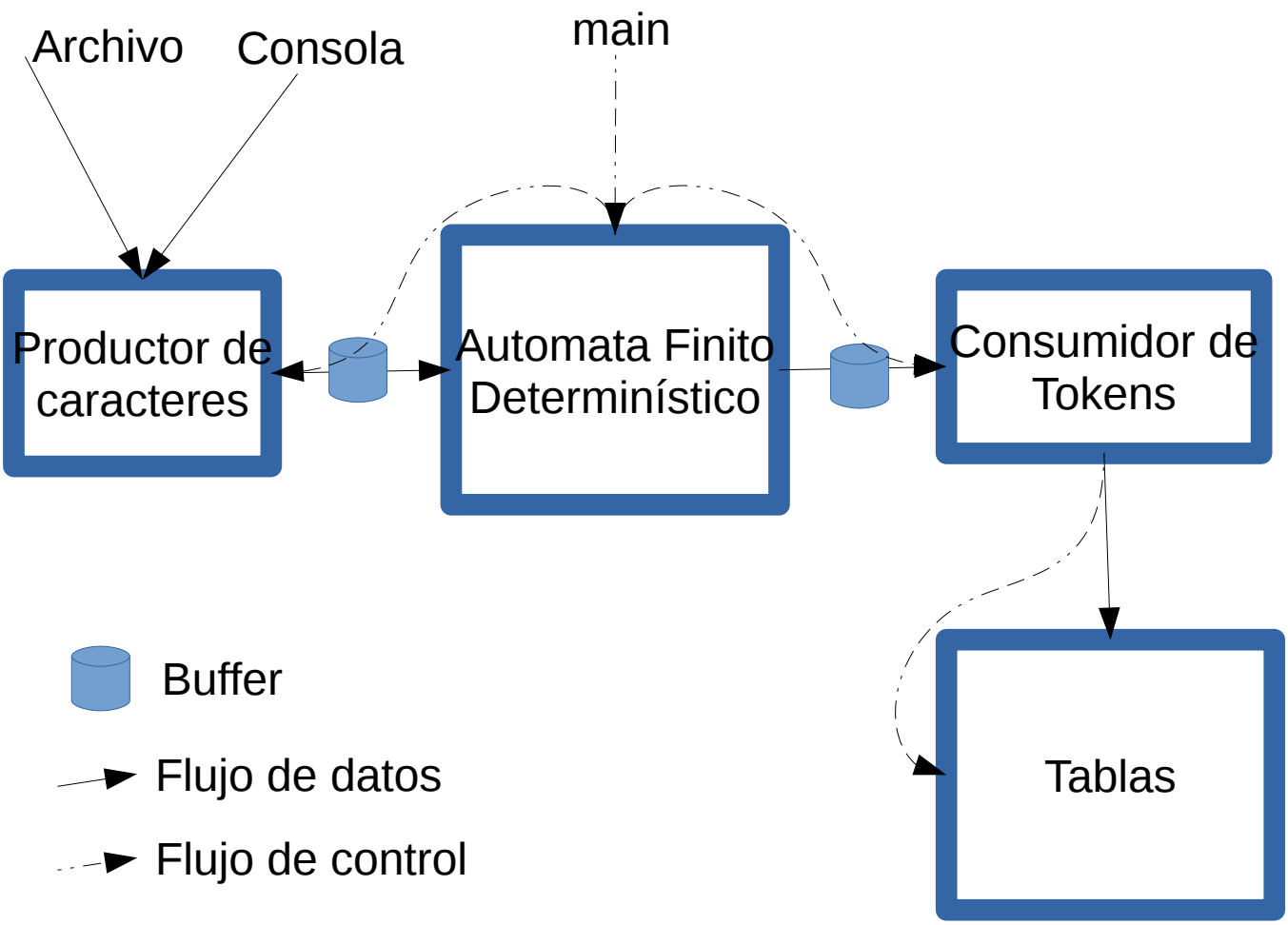



Componentes

**AFD**

- Lee de un buffer los caracteres.
- Hace peticiones para rellenar el buffer.
- Genera tokens y hace peticiones de salida al consumidor de tokens.

**Consumidor de Tokens**

- Recibe parámetros de los tokens (copia).
- Crea las entradas en las tablas si es necesario.
- Realiza el envío del buffer a la siguiente etapa de procesamiento.

**Productor de caracteres**

- Lee la entrada especificada y rellena el buffer interno.

**Contexto**

Almacena lista de identificadores y cadenas.

**Plan a implementar**

- Productor de caracteres que lee asíncronamente el archivo para evitar bloqueo.
- Soporte de caracteres multi-byte.

**Análisis de requerimientos**

- Los enunciados en el documento.
    - Leer y generar archivos
    - Identificar tokens
    - Generar tablas
    - Mostrar errores
- Paralelismo.
    - Grueso: Lexer no usa estado global y puede haber varias instancias.
    - Medio: Pipelining entre las etapas. Thread Pouling (posible)

Flex usa estado global y se necesitaría editar manualmente cada vez que se modifique la gramática regular para habilitarlo en consideraciones paralelas.

Viabilidad de escribir manualmente un lexer.

Ventajas:

- A la medida.
- Es posible simular transiciones -E.
- La comunicación entre componentes se puede implementar a gusto. (callbacks, mutex, llamadas directas, modo de sincronzación)

Desventajas:

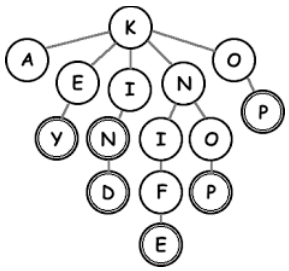
- Tiempo de desarrollo y debugging.
- Puede resultar algo inmantenible cuando los token se vuelven largos y semejantes.

Solución: Escribir un pequeño preprocesador muestra que pueda generar el código de las cadenas (identificadores, palabras reservadas, tipos, etc.)

- Este tipo de expresiones largas son las que añadían mucha complejidad al AFD.
- Se reutiliza conocimiento y código de la plataforma escogida C++.
- El archivo puede quedar listo sin edición manual.

El preprocesador reconoce las siguentes expresiones

| | |
|---|---|
| @ | 1ª aparicion: lugar donde se insertará el codigo generado.<br>2ª aparición: lugar de donde se leen las cadenas a reconocer. |
| (Σ+)$ | Tipos base |
| (Σ+)# | Funciones |
| (Σ+)! | Procedimientos |
| (Σ+) | Palabras reservadas |




Para realizar su función, el preprocesador crea un arbol trie: una estructura ligada, con multiples hijos por nodo, en el cual la cadena se almacena como el trayecto del nodo base a las hojas.

Luego el trie se recorre por profundidad, mientras se genera el código para cada estado de la FSM: cada nodo hijo representa una transicion y las hojas son los estados aceptores. Además se genera un estado para las cadenas que no cumplen con ninguna de las palabras del lenguage, pero que esten constituidos de letras, para que formen los identificadores.

El preprocesador usa de macros que deben ser implementados por el archivo fuente, ademas de un estado default, que es el punto de entrada de la máquina de estados.

El sistema acepta el siguiente lenguage compuesto de los siguientes elementos.

| Expresión | Clase |
|---|---|
| *( ) \| ; \| , \| : \| . \| [ ]* | Cáracteres especiales |
| *> \| >= \| <= \| <> \| =* | Operadores relacionales |
| *AND \| ARRAY \| BEGIN \| DIV \| DO \| DOWNTO \|ELSE \| END \| FOR \| IF \| MOD \| NOT OR \| PROGRAM \| REPREAT \| THEN \| TO \| UNTIL \| VAR \| WHILE* | Palabras reservadas |

| | |
|---|---|
| *boolean* \| *char* \| *integer* \| *real* \| *text* | Tipos base |
| *get* \| *put* \| *read* \| *readln* \| *write* \| *writeln* | Procedimientos |
| *abs* \| *chr* \| *cos* \| *eof* \| *eoln* \| *exp* \| *ln* \| *sin* \| *sqr* \| *sqrt* \| *trunc* | Funciones |
| [*a-Z*] ([*a-Z*] \| [*0-9*])* | Identificador |
| *0* \|<br>[*1-9*] [*0-9*]* \|<br>*0* [*0-7* ]* \|<br>*0x* [*0-9*] ([*a-f*] \| [*A-F*])* *0* \|<br>[*1-9*] [*0-9*]* | Literal entero |
| (*0* \| [*1-9*] [*0-9*]*) . [*0-9*]+ | Literal flotante |
| *:=* | Asignación |
| *+* \| */* \| ** \| *-* | Operadores aritméticos |

La lista de token procesados se implementó con un vector (arreglo dinámicamente asignado) y las tablas se implemntaron como tablas de hash.