INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

Compiladores

Alumnos:
García Gasca Axel Gabriel
Taboada Montiel Enrique

Profesora:

Albortante Morato Cecilia

Práctica 2 – Analizador Léxico

Grupo: 5CM4

Introduccion

Un analizador léxico, también conocido como scanner o tokenizador, es una de las primeras fases del proceso de compilación o interpretación de un lenguaje de programación. Su objetivo principal es tomar una secuencia de caracteres (es decir, el código fuente) y dividirla en componentes más pequeños llamados tokens, que son las unidades mínimas de significado para un lenguaje de programación. Estos tokens se pasan luego a la siguiente fase del compilador o intérprete, que normalmente es el analizador sintáctico*o parser.

Un analizador léxico lee el código fuente carácter por carácter, formando y reconociendo secuencias válidas (tokens) de acuerdo con las reglas de un lenguaje. Los tokens pueden ser palabras clave, identificadores, operadores, números, símbolos especiales, etc. Por ejemplo, en el lenguaje de programación C, el código `int x = 10;` podría descomponerse en los siguientes tokens:

- `int`: palabra clave
- `x`: identificador
- `=`: operador de asignación
- `10`: número entero
- `;`: símbolo especial

Componentes básicos de un analizador léxico

1. Tabla de símbolos: Una estructura de datos donde se almacenan identificadores y palabras clave, permitiendo distinguir entre ellos cuando se encuentran en el código fuente.

2. Autómata finito: La mayoría de los analizadores léxicos se construyen utilizando autómatas finitos deterministas (AFD), que permiten reconocer patrones de cadenas (como palabras clave o números) mediante un conjunto de estados y transiciones.

3. Tokens: Los resultados que devuelve el analizador léxico. Cada token tiene un tipo (p. ej., identificador, número, operador) y un valor asociado (p. ej., el nombre del identificador o el valor del número).

Proceso de construcción de un analizador léxico

1. Definir el alfabeto y los tokens: El primer paso es identificar qué símbolos forman parte del alfabeto (p. ej., letras, dígitos, operadores) y los tokens válidos que deben reconocerse (p. ej., palabras clave, números, operadores).

2. Construir las reglas léxicas: Estas reglas se definen mediante expresiones regulares o diagramas de estados que describen cómo reconocer tokens. Por ejemplo:

- Identificadores: cualquier secuencia de letras y dígitos que comience con una letra.
- Números: una secuencia de dígitos.
- Operadores: símbolos como `+`, `-`, `*`, `/`.

3. Implementar el autómata finito: El analizador léxico se puede modelar como un autómata finito determinista (AFD), donde:

- Los estados representan la etapa actual de reconocimiento.
- Las transiciones representan el paso de un estado a otro basado en el carácter de entrada.
- Los estados finales indican que se ha reconocido un token completo.

4. Gestionar los errores: Es crucial que el analizador léxico tenga mecanismos para detectar secuencias de caracteres que no pertenecen a ningún token válido. Esto es conocido como **manejo de errores léxicos**.

Desarrollo

El constructor __init__ inicializa las estructuras necesarias para un automáta: un diccionario de transiciones, un conjunto de estados, el alfabeto, el estado inicial, los estados de aceptación y el estado actual.

```
def __init__(self, archivo_automata):
    self.transiciones = {}   # Diccionario para almacenar las transiciones
    self.estados = set()     # Conjunto Q de estados
    self.alfabeto = set()    # Alfabeto Σ
    self.estado_inicial = None  # Estado inicial q0
    self.estados_aceptacion = set()  # Conjunto F de estados de aceptación
    self.estado_actual = None  # Estado actual del autómata
    self.cargar_automata(archivo_automata)
```

Luego llama a cargar automata para leer la configuración desde un archivo, el archivo contiene toda la configuración del automáta. Es importante mencionar que a partir de la quinta línea, carga las transiciones. Cada transición tiene un estado actual, una entrada y un estado siguiente. Estas transiciones se almacenan en el diccionario self.transiciones

```
def cargar_automata(self, archivo_automata):
    with open(archivo_automata, 'r') as archivo:
        lineas = archivo.readlines()

        # Cargar conjunto de estados Q
        self.estados = set(lineas[0].strip().split(','))
        print(f"Estados cargados: {self.estados}")  # Depuración: imprimir estados

        # Cargar alfabeto Σ
        self.alfabeto = set(lineas[1].strip().split(','))
        print(f"Alfabeto cargado: {self.alfabeto}")  # Depuración: imprimir alfabeto

        # Cargar estado inicial q0
        self.estado_inicial = lineas[2].strip()
        print(f"Estado inicial: {self.estado_inicial}")  # Depuración: imprimir estado inicial

        # Cargar conjunto de estados de aceptación F
        self.estados_aceptacion = set(lineas[3].strip().split(','))
        print(f"Estados de aceptación: {self.estados_aceptacion}")  # Depuración: imprimir estados de aceptac
```

```
26
27              # Cargar conjunto de estados de aceptación F
28              self.estados_aceptacion = set(lineas[3].strip().split(','))
29              print(f"Estados de aceptación: {self.estados_aceptacion}")  # Depuración: imprimir estados de acept
30
31              # Cargar las transiciones δ
32              for linea in lineas[4:]:
33                  estado_actual, entrada, estado_siguiente = linea.strip().split(',')
34                  if estado_actual not in self.transiciones:
35                      self.transiciones[estado_actual] = {}
36                  self.transiciones[estado_actual][entrada] = estado_siguiente
37
38      # Inicializamos la estructura de tokens como un diccionario de diccionarios de listas
```

También se define un diccionario global tokens, que almacenará las cadenas procesadas, clasificadas por tipo

```
# Inicializamos la estructura de tokens como un diccionario de diccionarios de listas
tokens = {}
```

Después tenemos la funcion que añade una cadena evaluada al diccionario de tokens, categorizándola bajo el tipo correspondiente.

```
# Función para agregar una cadena evaluada a la estructura de tokens
def agregar_token(tipoCadenaAnterior, cadena_evaluada):
    # Si el tipoCadenaAnterior no está en el diccionario del estado actual, lo creamos
    if tipoCadenaAnterior not in tokens:
        tokens[tipoCadenaAnterior] = []

    # Añadimos la cadena evaluada a la lista correspondiente
    tokens[tipoCadenaAnterior].append(cadena_evaluada)
```

La funcion transición maneja las transiciones del automata. Dado un carácter de entrada, verifica si existe una transición válida y cambia el estado del automáta en consecuencia. También clasifica la entrada como número, letra, operador, etc.

```python
def transicion(Automata, caracter):
    estado_str = str(Automata.estado_actual)  # Convierte el estado actual a una cadena

    # Clasificación de entradas para manejar dígitos, letras y símbolos especiales
    if caracter.isdigit():  # Si el carácter es un dígito, tratamos la entrada como 'digit'
        entrada = 'numeros'
    elif caracter.isalpha():  # Si el carácter es una letra
        entrada = 'letras'
    elif caracter == '+' or caracter == '-' or caracter == '/' or caracter == '*' :
        entrada = 'operadores'
    else:
        entrada = caracter  # Otros caracteres posibles

    if(estado_str == ' '):
        return 3
```

```python
def transicion(Automata, caracter):
    if(estado_str == ' '):
        return 3


    if estado_str in Automata.transiciones and entrada in Automata.transiciones[estado_str]:  # Verifica si exist
        nuevo_estado = Automata.transiciones[estado_str][entrada]
        Automata.estado_actual = nuevo_estado
        if(entrada == 'letras'):
            entrada = 'identificadores'
        return entrada
    else:
        estado_str = str(Automata.estado_inicial)
        if estado_str in Automata.transiciones and entrada in Automata.transiciones[estado_str]:  # Verifica si e
            nuevo_estado = Automata.transiciones[estado_str][entrada]
            Automata.estado_actual = nuevo_estado
            if(entrada == 'letras'):
                entrada = 'identificadores'
            return [2, entrada]
```

La funcion de inicio procesa la cadena de entrada carácter por carácter, realizando transiciones en el automáta y almacenando las cadenas evaluadas (tokens) según el tipo. Reinicia el automáta cuando se completa un token y lo almacena usando agregar_token.

```python
def inicio(Automata, cadena):
    Automata.estado_actual = Automata.estado_inicial  # Reiniciamos al estado inicial
    cadena_evaluada = ''
    tipoCadenaAnterior = ''
    tipoCadenaActual = ''

    #evaluar cada caracter
    for c in cadena:
        #guardar lo obtenido
        resultado = transicion(Automata,c)
        #si es una cadena se guarda el caracter actual en la cadena y el tipo de cadena

        if(isinstance(resultado, str)):
            if(Automata.estado_actual in Automata.estados_aceptacion):
                cadena_evaluada += c
            tipoCadenaActual = resultado

        #si es una cadena se guarda en el tipo de cadena
        elif(isinstance(resultado, list)):
            if(Automata.estado_actual in Automata.estados_aceptacion):
                agregar_token(tipoCadenaAnterior, cadena_evaluada)
            cadena_evaluada = c
            tipoCadenaActual = resultado[1]


        #si es cadena vacia el resultado que priviene
        elif(resultado == 3):
            if(cadena_evaluada != ''):
                if(Automata.estado_actual in Automata.estados_aceptacion):
                    agregar_token(tipoCadenaActual, cadena_evaluada)
            Automata.estado_actual = Automata.estado_inicial
            cadena_evaluada = ''
            tipoCadenaActual = ''
```

```python
        #si es cadena vacia el resultado que priviene
        elif(resultado == 3):
            if(cadena_evaluada != ''):
                if(Automata.estado_actual in Automata.estados_aceptacion):
                    adena_evaluada)
                    inicial
            cadena_evaluada = ''
            tipoCadenaActual = ''

        tipoCadenaAnterior =  tipoCadenaActual
    if(Automata.estado_actual in Automata.estados_aceptacion):
        agregar_token(tipoCadenaActual, cadena_evaluada)
```

(variable) cadena_evaluada: Literal['']

Resultados

Ejemplo con números

```
PS C:\Users\domo_> & C:/Users/domo_/AppData/Local/Microsoft/WindowsA
Estados cargados: {'5', '1', '3', '4', '6', '2', '0'}
Alfabeto cargado: {'', 'numeros', 'letras', 'operadores', ' '}
Estado inicial: 0
Estados de aceptación: {'1', '3', '2', '5'}
Ingrese la cadena: 5466774
Tipo 'numeros':
  -> 5466774
PS C:\Users\domo_>
```

```
PS C:\Users\domo_> & C:/Users/domo_/AppData/Local/Microsoft/WindowsApps/python3.10.exe
Estados cargados: {'1', '2', '4', '3', '0', '6', '5'}
Alfabeto cargado: {'letras', '', ' ', 'operadores', 'numeros'}
Estado inicial: 0
Estados de aceptación: {'1', '5', '2', '3'}
Ingrese la cadena: holamundo
Tipo 'identificadores':
  -> ho
  -> am
PS C:\Users\domo_>
```

Conclusión

Al realizar esta práctica reforzamos y aprendimos más sobre el analizador léxico, ya que es un elemento importante de un compilador, que su función es reconocer números, cadenas, palabras propias del lenguaje.

Continuamos avanzando para poder crear nuestro propio compilador y ya tenemos una parte importante de éste mismo, gracias a la práctica realizada y las diferentes clases que se impartieron en el aula de clase.

Referencias

Marciszack, M. M. (s.f.). *Máquinas de Estados: Autómatas Finitos Deterministas (AFD)*. Obtenido de GHD: https://www.institucional.frc.utn.edu.ar/sistemas/ghd/T-M-AFD.htm