Memoria Práctica PDL

Andrés Ollero Morales, Gabriel de Oliveira Trindade, Víctor Alejandro Sanz Ararat

Grupo 17

Índice

1. Diseño del Analizador Léxico

1.1. Diseño de Tokens

<boolean, >	<input, >	<print, >
<break, >	<int, >	<return, >
<case, >	<let, >	<string, >
<function, >	<switch, >	<if, >
<suma, >	<puntoComa, >	<asignacion, >
<negacion, >	<coma, >	<dosPuntos, >
<abrePar, >	<cierraPar, >	<comparacion, >
<abreLlave, >	<cierraLlave, >	<asignacionResto, >
<id, nºTS>	<constEnt, nº>	<cadena, "lexema">

1.2. Gramática

```
S -> \A | dB | %C | =D | _T | cT | { | } | ( | ) | + | : | ; | , | !
T -> cT | O.C
A -> lA | \
B -> dB | O.C
C -> =
D -> = | O.C

donde c = caracteres (a-z, A-Z), d = dígitos (0-9), l = cualquier cosa
y O.C = otro caracter distinto
```

1.3. Autómata

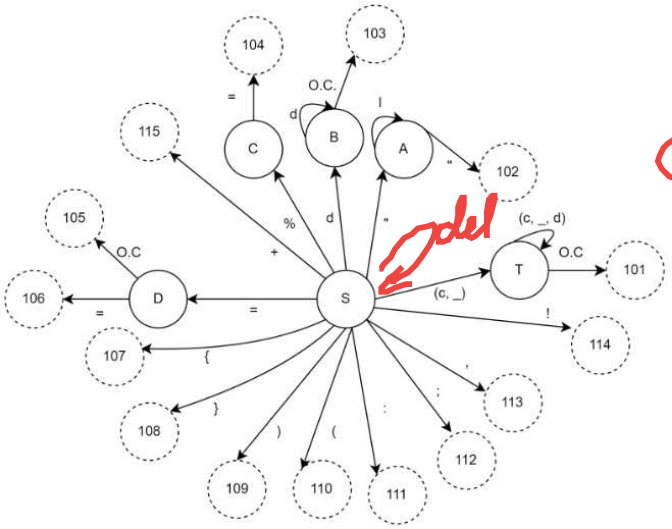

Figura 1: Implementación de la gramática con el autómata.

1.4. Acciones Semánticas

```
Se realiza la acción semántica LeerSigCaracter en todas las transiciones
menos en las transiciones:
    T : 101
    B : 103
    D : 105

S : T -> lexema = l; contador = 1;
T : T -> lexema = lexema  l ; contador++
T : 101 -> if (BuscarPalabraReservada(lexema) == 1)
               then GenerarToken(palabraReservada, lexema);
           else if (BuscarTablaSimbolos(lexema) == 1)
               then GenerarToken(id, posTS);
           else (BuscarTablaSimbolos(lexema) == 0)
               then AñadirEntradaTabla(id, posTS) &&
               GenerarToken(palabraReservada, posTS) && posTS++;
S : A -> contador = 0; lexema = \t
A : A -> lexema = lexema  l ; contador++;
A : 102 -> if (contador <= 64)
               then GT(lexema, cadena);
        else
            then printf(\Error, longitud de la cadena excede el limite");
S : B -> valor = valor_ascii(d);
B : B -> valor = valor + valor_ascii(d);
B : 103 -> if (valor <= 32767)
               then GT (valor, int)
        else
            then Printf(\Error, valor del entero excede el limite")
C : 104 -> if (c != '=')
               then Printf(\Error : Expected '%=")
           else
               GenerarToken(asignacionResto, )
D : 105 -> GenerarToken(igual, );
D : 106 -> GenerarToken(comparación, );
S : 107 -> GenerarToken(abreLlave, );
S : 108 -> GenerarToken(cierraLlave, );
S : 109 -> GenerarToken(abrePar, );
S : 110 -> GenerarToken(cierraPar, );
S : 111 -> GenerarToken(dosPuntos, );
S : 112 -> GenerarToken(puntoComa, );
S : 113 -> GenerarToken(coma , );
S : 114 -> GenerarToken(negación, );
S : 115 -> GenerarToken(suma, );

- GenerarToken (atributo, valor): Genera el token en un fichero de
                                  salida tokens.txt.
- BuscarPalabraReservada (lexema): Busca la palabra reservada en la
                                   respectiva tabla de palabras reservada.
```

4.1

```
- AñadirEntradaTabla(lexema, posTS): Añadimos el nombre de la variable a
                                     la tabla de símbolos.
```

1.5. Errores

Cuando se introduce en el código un símbolo que no pertenece al lenguaje, nuestro autómata lo omite. Los errores que contempla el código son cuando (y su correspondiente mensaje por línea de comandos):

- Se introduce un número superior a 32767
  Error: el valor numerico introducido excede el limite de 32767.
- Se intenta crear una cadena de más de 64 caracteres
  Error: Cadena sobrepasa los 64 caracteres.
- Se crea una cadena sin finalizar con la doble comilla final
  Error: La string esta mal formada.
- Se introduce exclusivamente un '%' sin el caracter '=' a continuación
  Error: Expected '=' after '%'.
- Se intenta hacer comentario de bloque y no se introducen los /* */
  Error: Comentario mal formado.

Y por supuesto, no se genera el token.

2. Diseño inicial de la Tabla de Símbolos

Cuando añadimos un identificador a la Tabla de Símbolos, primero comprobamos que no haya sido introducido previamente, y sino lo metemos. En el atributo del identificador pondremos el número que ha sido asignado, que en nuestro caso es el tamaño de la tabla en ese momento.

Decidimos que la mejor forma de implementar la Tabla de Símbolos era con la clase HashMap, que nos proporciona una búsqueda con complejidad constante, que a la hora de código con muchos identificadores la eficacia aumenta. El método de inserción para la clase es aleatorio por construcción, por tanto no tiene orden interno.

3. Casos de Prueba

3.1. Prueba Funcional 1

Código fuente:

```
1 int a = 42;
2 if(22 == 42)
3 {
4 let SSSadena = "hola mundo";
5 if (cadena == "hola mundo")
6 return 42;
7 }
```

Fichero de Tokens:

```
><palabraReservada, int>
<id, 1>
<igual, >
<constEnt, 42>
<puntoComa, >
<palabraReservada, if>
<abrePar, >
<constEnt, 22>
<comparacion, >
<constEnt, 42>
<cierraPar, >
<abreLlave, >
<palabraReservada, let>
<id, 2>
<igual, >
<cadena, "hola mundo">
<puntoComa, >
<palabraReservada, if>
<abrePar, >
<id, 3>
<comparacion, >
<cadena, "hola mundo">
<cierraPar, >
<palabraReservada, return>
<constEnt, 42>
<puntoComa, >
<cierraLlave, >
```

Tabla de Símbolos:

```
#0:
* lexema: 'a'
* lexema: 'cadena'
* lexema: 'SSSadena'
```

Como podemos ver en este caso el analizador léxico logra generar bien los tokens del fichero, saltándose las líneas en blanco, y generando los id correspondientes a las variables que va encontrando a lo largo del fichero: como ocurre con el entero 'a', la variable "cadena" y la variable "SSSadena". A su vez va generando los identificadores en la tabla de símbolos correctamente.

3.2. Prueba Funcional 2

Código fuente:

```
/*
Codigo generado por el equipo 42
Versi n 1.2.3
Codigo de prueba numero 3
*/
let cadena_4;
switch (cadena_4)
case "numero1":
return (1);
/*
A partir de aqui hay que agregar....
*/
case "numero2":
return (2);
/*
fin del fichero
*/
```

Fichero de Tokens:

```
<palabraReservada, let>
<id, 1>
<puntoComa, >
<palabraReservada, switch>
<abrePar, >
<id, 0>
<cierraPar, >
<palabraReservada, case>
<cadena, "numero1">
<dosPuntos, >
<palabraReservada, return>
<abrePar, >
<constEnt, 1>
<cierraPar, >
<puntoComa, >
<palabraReservada, case>
<cadena, "numero2">
<dosPuntos, >
<palabraReservada, return>
<abrePar, ><constEnt, 2>
<cierraPar, >
<puntoComa, >
```

Tabla de Símbolos:

```
#0:
* lexema: 'cadena_4'
```

En este otro ejemplo podemos ver que funciona perfectamente teniendo comentarios de bloque tanto al inicio, al final y entre medias del código, a su vez que comprobamos que se generan bien los tokens del switch case. A su vez podemos comprobar que se generó bien la tabla de símbolos.

3.3. Prueba Funcional 3

Código fuente:

```
function numero2 int (int x){
if (  (x == 2))
return "resultado 2";
break;
input x;
}
```

Fichero de Tokens:

```
<palabraReservada, function>
<id, 1>
<palabraReservada, int>
<abrePar, >
<palabraReservada, int>
<id, 2>
<cierraPar, >
<abreLlave, >
<palabraReservada, if>
<abrePar, >
<negacion, >
<abrePar, >
<id, 1>
<comparacion, >
```

```
<constEnt, 2>
<cierraPar, >
<cierraPar, >
<palabraReservada, return>
<cadena, "resultado 2">
<puntoComa, >
<palabraReservada, break>
<puntoComa, >
<palabraReservada, input>
<id, 1>
<puntoComa, ><cierraLlave, >
```

Tabla de Símbolos:

```
#0:
* lexema: 'numero2'
* lexema: 'x'
```

En este caso podemos observar el correcto funcionamiento de las palabras reservadas function, break y input. Además de poder ver el correcto guardado de los identificadores de la función en la tabla de símbolos, y la generación de tokens de negación.

3.4. Prueba No Funcional 1

Código fuente:

```
let a = 4;
if (a == 64){
    a%2;
}
switch(a){
    case 1:
        let asda;
        break;
}
```

Fichero de Tokens:

```
<palabraReservada, let>
<id, 1>
<igual, >
<constEnt, 4>
<puntoComa, >
<palabraReservada, if>
<abrePar, >
<id, 0>
<comparacion, >
<constEnt, 64>
<cierraPar, >
<abreLlave, >
<id, 0>
<constEnt, 2>
<puntoComa, >
<cierraLlave, >
<palabraReservada, switch>
<abrePar, >
<id, 0>
<cierraPar, >
<abreLlave, >
<palabraReservada, case>
<constEnt, 1>
<dosPuntos, >
```

```
<palabraReservada, let>
<id, 2>
<puntoComa, >
<palabraReservada, break>
<puntoComa, >
<cierraLlave, >
```

Tabla de Símbolos:

```
#0:
* lexema: 'a'
* lexema: 'asda'
```

En este ejemplo comprobamos si se detecta el error en la asignación con resto, ya que el % debe llevar después un =, si no, el automata no lo reconoce.

3.5. Prueba No Funcional 2

9.1

Código fuente:

```
let cuatro = 321;
if (cuatro = 3123123313123)
{
    switch (cuatro){
        case 4:
            break;
    }
}
```

Fichero de Tokens:

```
<palabraReservada, let>
<id, 1>
<igual, >
<constEnt, 321>
<puntoComa, >
<palabraReservada, if>
<abrePar, >
<id, 0>
<igual, >
<cierraPar, >
<abreLlave, >
<palabraReservada, switch>
<abrePar, >
<id, 0>
<cierraPar, >
<abreLlave, >
<palabraReservada, case>
<constEnt, 4>
<dosPuntos, >
<palabraReservada, break>
<puntoComa, >
<cierraLlave, >
<cierraLlave, >
```

Tabla de Símbolos:

```
#0:
* lexema: 'cuatro'
```

Aquí comprobamos que la cadena numérica introducida no supere el valor máximo de 32767.

3.6. Prueba No Funcional 3

Código fuente:

```
let cadena = "Esta cadena es mas grande de lo permitido por el analizador
    sintactico."
```

Fichero de Tokens:

```
<palabraReservada, let>
<id, 1>
<igual, >
```

Tabla de Símbolos:

```
#0:
* lexema: 'cadena'
```

En este código vemos como trata el analizador léxico una cadena que sobrepasa el máximo permitido (64), se puede observar que no genera el token cadena.

Índice de comentarios