Memoria

Nombre:
Cesar Jordy Arrese Huamán
Numero matricula:
W140237

Analizador:

El Análisis de mi programa consta de dos partes el Análisis léxico por una parte y el sintáctico-semántico por otra.

En particular el programa lee un fichero .js llamado “testeo.js” y vuelca un resultado por pantalla con el valor del análisis si es este es correcto no, para pasar posteriormente a imprimir un fichero “lexicotokens” para los tokens, otro fichero llamado ”TablaSimbolos” para la tabla de símbolos, un fichero “Parse” con lo que se comprobara que genera el árbol en el vaSt y por ultimo un fichero con el primer error encontrado llamado “Errores”.

Para el fichero de tokens se paso a utilizar un formato con palabras reservadas para el if ,do,while,write,etc y con un id al que se le pasa el puntero con su posición en la tabla de símbolos .

En cuanto el fichero de la tabla de símbolos se utiliza un lenguaje compacto para especificar cada entrada con el nombre tipo de variable y si es función el numero de parámetros, tipo de cada parámetro.

Por otro lado el fichero parse sigue la lógica del analizador sintáctico descendente por tablas que este programa sigue para verificar el archivo.js.

Analizador léxico:

Para realizar el analizador léxico he utilizado jflex. Con un archivo .lex llamado “lenguaje.lex” le pasamos el lenguaje que utilizamos para el programa y este los transforma en un fichero (“AnalizadorLexico.java”) que sirve para transformar los distintos tokens en objetos enum (“Terminales”). Todo esto es invocado por la clase tester.java que se encarga de realizar todas las operaciones tanto del léxico como del sintáctico-semántico.

Se ha tenido en cuenta que el token que sea un numero entero no sea mayor de 32767 y que los token cadena no incluya los caracteres comillas en el momento de pasar el token.
Por otro lado sino cumple con los requisitos establecidos en el.

Código usado para el lenguaje:

```
SALTO=[\r|\n]
ID = [a-z|A-Z][a-z|A-Z|0-9|_]*
ENTERO = 0|[1-9][0-9]*
CADENA=\"[a-z|A-Z|0-9|_|\s]*\"
COMENTARIO=([/][/][^\r|\n]*|[/][*][^*]*[*][/]|[ |\t|\f])
%%
"if" { return IF; }
"while" {return WHILE; }
"do" {return DO; }
"function" { return FUNCTION;}
"return" { return RETURN;}
"var" { return VAR;}
"int" { return INT;}
"chars" {return CHARS;}
"bool" {return BOOL;}
"write" {return WRITE;}
"prompt" {return PROMPT;}
"(" {return PARENIZ;}
")" {return PARENDE;}
"{" {return LLAVEIZ;}
"}" {return LLAVEDE;}
"," {return COMA;}
"==" {return IGUAL;}
"=" {return ASSIGN;}
"&=" {return ASSIGNY;}
"&&" {return AND;}
"!=" {return NOIGUAL;}
"<" {return MENOR;}
">" {return MAYOR;}
"+" {return SUMA;}
"-" {return RESTA;}
"*" {return MULT;}
{SALTO} {return SALTO;}
{ID} { return ID; }
{ENTERO} { return ENT; }
{CADENA} {return CAD; }
{COMENTARIO} {return COMENTARIO;}
"/" {return DIV;}
. { return ERROR; }
```

Analizado Sintáctico-Semántico

Para esta parte utiliza los tokens que venían del léxico y se los pasa a la clase “OperadoSemantico.java” que utiliza una pila p y una pila aux. para la realización del analizador semántico.

El funcionamiento es simple “OperadorSemantico.java” recibe a través de “realizaroperacion” el Terminal o token siguiente el cual utiliza para buscar los distintos consecuentes que este tiene cuando es obtenido a partir del antecedente que se encuentre en la cima de p que se descarta y pasa a la pila de aux. y el token al que quiere llegar. En caso de que no tenga consecuentes se pasara a marcar como error sintáctico.

Lo anterior solo es el caso de que la cima de p sea un no terminal, pero si es un terminal también se descarta y pasa a la cima de aux. pero antes se comprueba de que el token que llego coincide con el token descartado ya que si no se estaría produciendo otro error sintáctico.

Estos consecuentes que sacamos a partir del token y de p están especificados en una clase llamada “AntescedentesS.java” donde el objeto es creado con el No Terminal que seria el antecedente y con una operación “buscar” que recibe de parámetro el token.

Por otra parte en “AntescedentesS.java” también se considera como consecuentes las acciones semánticas necesarias para realizar las distintas comprobaciones de tipo.

Estas comprobaciones de tipo así como la administración de la Tabla de símbolos se produce en la clase “OperadorSemantico.java”. Esta comprobaciones se hace uso de una clase ConsecuenteS donde almacenas tanto terminales, como no terminales y acciones semánticas con el fin de poder tenerlos todas en una sola pila p y poder al mismo tiempo diferenciarlas. Aquí claramente es importante el uso de una pila aux . para pasar el tipo entre los distintos niveles del árbol sintáctico creado indirectamente a través del programa.

Por otro lado en la clase OperadorSemantico.java hay dos variables lógicas globales que se encargan de devolver el error siendo una resultadoSemantico y la otra resultadoSintactico.

Analizador Sintáctico – vaSt

Para la creación del árbol sintáctico se uso las siguientes producciones que coincide también con gramática usado en la clase AntescedentesS.java para que no genere problemas a la hora de hallar el árbol con el parse obtenido por el programa.

Este parse obtenido por el programa es generado de la misma forma que se genera para el analizador sintáctico-semántico que se plantea para conseguir su funcionalidad.

La clase encargada de generar cada numero del parse es AntescedentesParse.java que se crea con la cima de la pila de p si esta es un No terminal al igual que la otra dependiendo del token que llega la operación buscar de la clase AntescedentesParse.java te devolverá un numero de parse u otro.

Siendo las producciones estas:

PP -> P
P -> B Z P
P -> F Z P
P -> Z P
P -> eof
B -> var T id
B -> if (E) S
B -> do { Z C } while (E)
B -> S
C -> B Z C
C -> lambda
T -> int
T -> chars
T -> bool
S -> return X
S -> id SP
S -> write (E)
S -> prompt (id)
SP -> = E
SP -> (L)
SP -> &= E

X -> E
X -> lambda
L -> E Q
L -> lambda
Q -> , E Q
Q -> lambda
E -> R EP
EP -> && R EP
EP -> lambda
R -> U RP
RP -> < U
RP -> > U
RP -> lambda
U -> V UP

UP -> + V UP
UP -> - V UP
UP -> lambda
V -> W VP
VP -> * W VP
VP -> / W VP
VP -> lambda
W -> id WP
W -> ent
W -> cad
W -> (E)
WP -> (L)
WP -> lambda
F -> function H id (A) Z { Z C }
H -> T
H -> lambda
A -> T id K
A -> lambda
K -> , T id K
K -> lambda
Z -> salto ZP
ZP -> salto ZP
ZP -> lambda

Modo de uso del programa:

El programa leerá el fichero .js que esta en la carpeta “procesadores2” el cual se podrá realizar a través de la clase tester.java Por otro lado en la misma carpeta se generaran todos los ficheros que se requieren.

Anexo:

Casos correctos

1)

En este caso se ha intentado comprobar que los returns de una función funcionan correctamente y que se creen correctamente en la TS.

código:

```
var chars id1
id1="hola"
function int id2(int x,chars y)
{
	write(y)
	var int a1
	a1=x*x
	return a1
}
function chars id3(int x)
{
	var chars b2
	b2=id1
	return b2
}
```

fichero tokens:

```
<PALRES,var>
<PALRES,chars>
<ID,0>
<SALTO,>
<ID,0>
<ASIGN,>
<CAD,hola>
<SALTO,>
<PALRES,function>
<PALRES,int>
<ID,1>
<PARENIZ,>
```

<PALRES,int>
<ID,0>
<COMA,>
<PALRES,chars>

<ID,1>
<PARENDE,>
<SALTO,>
<LLAVEIZ,>
<SALTO,>
<PALRES,write>
<PARENIZ,>
<ID,1>
<PARENDE,>
<SALTO,>
<PALRES,var>
<PALRES,int>
<ID,2>
<SALTO,>
<ID,2>
<ASIGN,>
<ID,0>
<MULT,>
<ID,0>
<SALTO,>
<PALRES,return>
<ID,2>
<SALTO,>
<LLAVEDE,>
<SALTO,>
<PALRES,function>
<PALRES,chars>
<ID,2>
<PARENIZ,>
<PALRES,int>
<ID,0>
<PARENDE,>
<SALTO,>
<LLAVEIZ,>
<SALTO,>
<PALRES,var>
<PALRES,chars>
<ID,1>

<SALTO,>
<ID,1>
<ASIGN,>
<ID,0>
<SALTO,>

<PALRES,return>
<ID,1>
<SALTO,>
<LLAVEDE,>
<SALTO,>
<EOF,>

Tabla de Símbolos

#1:
*'id1'
+tipo:'chars'
*'id2'
+tipo:'int'
+parametros:2
+tipoparam1:'int'
+tipoparam2:'chars'
+idtabla:2
*'id3'
+tipo:'chars'
+parametros:1
+tipoparam1:'int'
+idtabla:3
#2:
*'x'
+tipo:'int'
*'y'
+tipo:'chars'
*'a1'
+tipo:'int'
#3:
*'x'
+tipo:'int'
*'b2'
+tipo:'chars'

Parse

Des 1 2 6 13 56 58 2 9 16 19 28 31 35 39 45 42 38 34 30 56 58 3 49 50 12 52 12 54 13 55 56 58 56 58 10 9 17 28 31 35 39 43 48 42 38 34 30 56 58 10 6 12 56 58 10 9 16 19 28 31 35 39 43 48 40 43 48 42 38 34 30 56 58 10 9 15 22 28 31 35 39 43 48 42 38 34 30 56 58 11 56 58 3 49 50 13 52 12 55 56 58 56 58 10 6 13 56 58 10 9 16 19 28 31 35 39 43 48 42 38 34 30 56 58 10 9 15 22 28 31 35 39 43 48 42 38 34 30 56 58 11 56 58 5

2)

Aquí se comprueba que la función sea recursiva y que también se pasen correctamente sus tipos tanto dentro como fuera de esta

código:

```
var int id1
id1=3
function int id2(int x)
{
	return id2(x+1)
}
var int id3
id3=id2(id1)
```

3)

Se comprueba de que funciona bien los tipos y la sentencias simples

código:

```
var bool id1
var int id3
id1=id3>1
function id2(bool x)
{
	if(x) id1 &= x
}

id2(id1)
```

4)

Se comprueba de que funciona el do-while

código:

```
var int id1
prompt(id1)
var int id3
id3=10
do{
	id3=id3-1
}while(id3>1)

function bool id2(int y)
{
	return id3<y
}
var bool id4
id4=id2(2)
```

5)

Se comprueba de que una función sin argumentos y sin tipo pueda no devolver nada

código:

```
var chars id1
var chars id2
id1="vacio"
function id3()
{
	id2=id1
}
id3()
```

Casos incorrectos:

1)

Se pasa un token incorrecto

código:

```
var int id1#
var chars id2
id1="vacio"
function id3()
{
	id2=id1
}
id3()
```

Salida:

```
Error Lexico
En linea: 1
```

2)

código:

```
var chars id1
var chars chars id2
id1="vacio"
function id3()
{
        id2=id1
}
id3()
```

Salida:

```
Error Sintactico
En linea: 2
```

3)

Se comprueba la relación de tipos

código:

```
var int id1
var chars id2
id1="vacio"
function id3()
{
        id2=id1
}
id3()
```

Salida:

```
Error Semantico
En linea: 3
```

4)

Se comprueba de que la se pase correctamente los argumentos a una función

código:

```
var chars id1
var chars id2
id1="vacio"
function id3(int x)
{
        id2=id1
}
id3()
```

Salida:

```
Error Semantico
En linea: 8
```

5)

Se comprueba de que la función sepa que cuando recibir un return

código:

```
var chars id1
var chars id2
id1="vacio"
function int id3(int x)
{
        id2=id1
}
id3(3)
```

Salida:

```
Error Semantico
En linea: 7
```