26-4-2021

JFlex y Cup

DOCUMENTACION PRACTICA 2

MALISZEWSKI, PIOTR
MARROQUÍN RIVAS, ANDERSON
NALEPA, MACIEJ
TEORÍA DE AUTÓMATAS Y COMPUTACIÓN [E05]

Componentes del Lenguaje

Expresiones Regulares Aritméticas		
“;” ’	**SEMICOLON**	Ej: for (i = 1; i< x; i++) …
	Símbolo terminador de línea/expresiones	
“ ” ’	**COMMA**	Ej: int a, b = 1;
	Símbolo separador de expresiones	
“=”	**ASSIGN**	Ej: a = 0
	Símbolo de asignación de expresiones	
“+”	**PLUS**	Ej: a + 2
	Símbolo sumador de elementos	
“-“	**MINUS**	Ej: a – 3
	Símbolo sustractor de elementos	
“*”	**MUL**	Ej: a*2
	Símbolo multiplicador de elementos	
“/”	**DIV**	Ej: a/2
	Símbolo particionado de un elemento con respecto a otro	

Expresiones Regulares Relacionales		
“<”	**LESS**	Ej: a < b
	Símbolo de comparación de elementos, devolverá verdadero en caso de que “a” sea menor con respecto de “b”.	
“<=”	**L_EQUAL**	Ej: a <= b
	Símbolo de comparación de elementos, devolverá verdadero en caso de que “a” sea mero que “b”.	
“>”	**GREATER**	Ej: a > b
	Símbolo de comparación de elementos, devolverá verdadero en caso de que a sea mayor que “b”.	
“>=”	**G_EQUAL**	Ej: a >= b
	Símbolo de comparación de elementos, devolverá verdadero en caso de que “a” sea mayor o igual a “b”.	
“==”	**EQUAL**	Ej: a == b
	Símbolo de comparación de elementos, devolverá verdadero en caso de que “a” sea estrictamente igual a “b”.	
“!=”	**NOT_EQUAL**	Ej: a != b
	Símbolo de comparación de elementos, devolverá verdadero en caso de que “a” sea distinto a “b”.	

Expresiones Regulares Lógicas		
“&&”	**AND**	Ej: (a > 0) && (a < 1)
	Símbolo que permite evaluar expresiones y/o variables, devolverá verdadero en caso de que las valoraciones de verdad de ambas expresiones sean verdaderas.	
	OR	Ej: (a == 0) || (a < 1)

“||”	Símbolo que permite evaluar expresiones y/o variables, devolverá verdadero en caso de que las valoraciones de verdad de al menos una expresiones sean verdaderas.	
“!”	**NOT**	Ej: !a Ej: !(a < 1)
	Símbolo que permite invertir el valor de verdad de una expresión y/o variable .	

Tipos de Expresiones Regulares		
“void”	**T_VOID**	Ej: public static void main (String args[]) {...}
	Tipo que indica que, el determinado método, no retorna valor alguno.	
“boolean”	**T_BOOL**	Ej: boolean a = true
	Tipo que indica que el contenido de una variable o el retorno de un método es un valor de verdad (true/false).	
“int”	**T_INT**	Ej: int a = 1
	Tipo que indica que el contenido de una variable o el retorno de un método es un valor entero.	

Tipos Funcionales		
“return”	**RETURN**	Ej: return a return true
	Sentencia que se emplea para interrumpir la secuencia de ejecución de las sentencias de un método y/o devolver un valor.	
“while”	**WHILE**	Ej: while (a < 10) ...
	Sentencia que se emplea para la declaración de un bucle, el cual que ejecutara mientras la condición de guarda se evalúe como verdadera.	
“for”	**FOR**	Ej: for (i = 1; i < x; i++) ...
	Sentencia que se emplea para la declaración de un bucle, el cual que ejecutara mientras la expresión interna se evalúe como verdadera. Puede implementar la asignación de una variable auxiliar y el patrón de incremento de esta.	
“class”	**CLASS**	Ej: class Lenguaje {...}
	Sentencia que se corresponde para la creación de estructuras tipo clase.	
“static“	**STATIC**	Ej: static int a = 2;
	Sentencia que se corresponde a la directiva homónima, a qué elementos se puede aplicar, sus características, beneficios y desventajas que pueden tener.	
“public”	**PUBLIC**	Ej: public class ABC {...}
	Corresponde al modificador de acceso homónimo, se puede aplicar a clases, métodos y variables. Indica que es accesibles para cualquier parte del código.	

Otros Componentes del Lenguaje		
“true”	**FLAG**	Ej:
	Expresión del lenguaje que se corresponde al valor de verdad “true”.	
“false”	**FLAG**	Ej:
	Expresión del lenguaje que se corresponde al valor de verdad “false”.	
“[a-zA-Z_] [0-9a-zA-Z_]*”	**IDENT**	Ej: {a, b, A, B, aAbB, a1, ...}
	Expresión regular que generan un lenguaje que contiene un numero infinito de palabras, que se corresponden al identificador de las variables.	
“[0-9]+”	**NUMBER**	Ej: {0, 1, 2, 3, ..., 10, ..., 7168, ...}
	Expresión que genera un lenguaje que contiene un numero infinito de palabras, que se corresponden a valores enteros sin signo.	

Explicación del Código JFlex

Definición de los elementos léxicos que componen el lenguaje que se ha implementado para la práctica. Cuando uno de los elementos es detectado por el programa, se devuelve el valor correspondiente según la etiqueta definida para ese elemento.

```
";" { return new Symbol(sym.SEMICOLON); }
"," { return new Symbol(sym.COMMA); }
"=" { return new Symbol(sym.ASSIGN); }
"+" { return new Symbol(sym.PLUS); }
"-" { return new Symbol(sym.MINUS); }
"*" { return new Symbol(sym.MUL); }
"/" { return new Symbol(sym.DIV); }
```

s elementos operandos aritméticos propuestos para el ıtos de las operaciones más elementales para acercar las

aje que se han identificados los mostrados a continuación, tenido en cuenta para su definición el conjunto se signos en la lectura (elemento definido "**==**", elemento no definido "**=**""**=**"). Se han identificado todas las principales relaciones entre elementos.

```
">" { return new Symbol(sym.GREATER); }
">=" { return new Symbol(sym.G_EQUAL); }
"<" { return new Symbol(sym.LESS); }
"<=" { return new Symbol(sym.L_EQUAL); }
"==" { return new Symbol(sym.EQUAL); }
"!=" { return new Symbol(sym.NOT_EQUAL); }
```

```
"&&" { return new Symbol(sym.AND); }
"||" { return new Symbol(sym.OR); }
"!" { return new Symbol(sym.NOT); }

"(" { return new Symbol(sym.PAREN_L); }
")" { return new Symbol(sym.PAREN_R); }
"{" { return new Symbol(sym.BRAC_L); }
"}" { return new Symbol(sym.BRAC_R); }
```

Se han tomado el "***y***", el "***o***", y el "***no***" lógicos como elementos operacionales para el lenguaje por su simplicidad y su gran presencia en estructura computacionales, siendo incorporados en los programas ***cláusulas de Horn*** y conjunciones de estas.

Así como también se definen los elementos de agrupamiento, como son los paréntesis y corchetes, para la agrupación de las cláusulas u otras composiciones de aritmeticológicas.

```
"void" { return new Symbol(sym.T_VOID); }
"boolean" { return new Symbol(sym.T_BOOL); }
"int" { return new Symbol(sym.T_INT); }

/* functional */
"return" { return new Symbol(sym.RETURN); }
"while" { return new Symbol(sym.WHILE); }
"for" { return new Symbol(sym.FOR); }
"class" { return new Symbol(sym.CLASS); }
"static" { return new Symbol(sym.STATIC); }
"public" { return new Symbol(sym.PUBLIC); }
```

Como tipos de parámetros para la variables y métodos se han identificado "***void***", "***boolean***" e "***int***".

Como sentencias funcionales se han implementado las siguientes, como por ejemplo "***return***" para métodos con retorno e parámetros, los bucles "***for***" y "***while***" por su versatilidad. También se han implementado "***class***" para creación del programa y estructuras. Y "***static***", como "***public***" para definir las características, así como para la fácil accesibilidad interna de las variables y métodos.

```
[a-zA-Z_][0-9a-zA-Z_]* { return new Symbol(sym.IDENT, yytext()); }
```

Para el identificador de las variables, se presenta la implementación de un leguaje que genera un numero infinito palabras que pueden o no empezar por mayúsculas y poder estar seguido de otro conjunto de valores alfanuméricos. Hecho esta para darle al leguaje un espectro de representación más amplio.

```
[0-9]+ { return new Symbol(sym.NUMBER, new Integer(yytext())); }
```

Para la definición de del conjunto de valores numéricos se a tomado un leguaje compuesto por un numero infinito de palabras que se corresponden al conjunto de todos los enteros sin signo, pues este se le asigna mediante la expresión matemática.

Explicación del Código Cup

El núcleo del programa, el método ***Yylex()***, lo ejecuta el método ***main()***. Dentro de dicho método, ...iediante un ***scanner()***. Tras la recepción ...le recursos.

```
parser code {:
        public static void main(String args[]) throws Exception {

                System.out.print("Inserte un nombre de archivo: ");
                Scanner scanner = new Scanner(System.in);
                String filename = scanner.next();
                scanner.close();

                FileInputStream stream = new java.io.FileInputStream(filename);
                Reader reader = new java.io.InputStreamReader(stream);

                try {
                        new parser(new Yylex(reader)).parse();
                }
                catch (Exception e) {
                        System.err.println("ERROR Analisis INCORRECTO !");
                        System.exit(1);
                }
                System.out.println("Analisis CORRECTO");
        }
:}
```

...eam()* para la apertura del archivo, y con ...

...os del lenguaje a ***Yylex()*** el que analizara ...excepción y se finaliza la ejecución. Si el ...on normalidad.

aritmeticológicos, etc.

Y se descartan aquellos que no nos interesan o no están definidos en el susodicho lenguaje (como ***no terminal***).

También definimos de los aritméticos (“(”, “)”, “{”, “}”) como **precedente** para determinar en qué posición estos pueden colocarse dentro del lenguaje.

```
...COMMA, // ; ,
...AREN_R, // ( )
...AC_R, // { }
 =
...BOOL, T_INT, // types
...ILE, FOR, CLASS, STATIC, PUBLIC, // functional
...S, MUL, DIV, // aritmethitical
        GREATER, G_EQUAL, LESS, L_EQUAL, EQUAL, NOT_EQUAL, // relational
        AND, OR, NOT; // logical
terminal String IDENT;
terminal Integer NUMBER;
terminal boolean FLAG;

non terminal compound, cmd, expr_part, expr_list;
non terminal var_expr, var_inc, var_assign, var_def;
non terminal func_call, func_args, arg;
non terminal math_i, math_expr, logical_i, logical_expr, relational_expr;
non terminal func_dec, class_dec, loop;

precedence left PLUS, MINUS;
precedence left MUL, DIV;
```

Las definiciones :

- ***cmd***: que se corresponde a la estructura que tendrá una inicialización completa de una variable de una función de retorno.
- ***compound***: que se corresponde a la estructura de la clase.
- ***var_def***: que se corresponde a la estructura de la línea de comandos con la que es invocado.
- ***math_i***: define la estructura de las variables con su valor, tipo e identificador, respectivamente.
- ***math_expr***: define la estructura de una expresión matemática valida.

```
cmd ::= var_inc | var_def | var_assign | func_call | RETURN var_expr;

compound ::= class_dec | func_dec | loop;

var_def ::= T_BOOL IDENT | T_INT IDENT;

math_i ::= NUMBER | func_call | IDENT;
math_expr ::= PAREN_L math_i PAREN_R //bug appears if math_i is changed to math_expr
        | math_i PLUS math_expr
        | math_i MINUS math_expr
        | math_i MUL math_expr
        | math_i DIV math_expr
        | UMINUS math_i;/*
        | math_i;*/ // The bug appears when math_i is added
```

```
69
70  var_expr ::= PAREN_L var_expr PAREN_R | math_expr | UMINUS math_expr;
71  var_assign ::= IDENT ASSIGN var_expr | var_def ASSIGN var_expr;
72  var_inc ::= IDENT PLUS PLUS | IDENT MINUS MINUS;
73
74  func_call ::= T_VOID IDENT PAREN_L func_args PAREN_R
75  | T_BOOL IDENT PAREN_L func_args PAREN_R
76          | T_INT IDENT PAREN_L func_args PAREN_R;
77  func_args ::= arg COMMA func_args | arg | ;
78  arg ::= T_VOID | T_VOID IDENT | T_BOOL IDENT | T_INT IDENT;
79
80  logical_i ::= NOT IDENT | IDENT;
81  relational_expr ::= PAREN_L relational_expr PAREN_R
82          | NOT PAREN_L relational_expr PAREN_R
83          | logical_i GREATER logical_i
84          | logical_i G_EQUAL logical_i
85          | logical_i LESS logical_i
86          | logical_i L_EQUAL logical_i
87          | logical_i EQUAL logical_i
88          | logical_i NOT_EQUAL logical_i;
89  logical_expr ::= relational_expr
90          | logical_expr AND relational_expr
91          | logical_expr OR relational_expr;
92
93  class_dec ::= PUBLIC CLASS IDENT;
94  func_dec ::= PUBLIC STATIC IDENT PAREN_L func_args PAREN_R;
95  loop ::= WHILE PAREN_L logical_expr PAREN_R
96          | FOR PAREN_L var_assign SEMICOLON logical_expr SEMICOLON var_expr PAREN_R;
97
```

- ***var_expr***: define la estructura valida que pueden tener las variables.
- ***var_assign:*** define la estructura valida que pueden tener las asignaciones de variables.
- ***var_inc:*** define la estructura valida que pueden tener las inicializaciones de variables.

A continuación se definen las expresiones creadas para las llamadas a funciones:

- ***func_call:*** declara la estructuras posibles para las llamadas de función válidas.
- ***func_args:*** indica la estructura, y cuáles son los argumentos que se les puede pasar a una función.
- ***args:*** indica los tipos que pueden tomar los argumentos válidos.
- ***logical_i:*** ???
- ***realtional_expr:*** indica la estructura valida de las operaciones relacionales entre elementos.
- ***logical_expr:*** indica la estructura valida de las operaciones lógicas.
- ***class_dec:*** indica la declaración valida de una clase pública.
- ***func_dec:*** indica la declaración valida de una función pública.
- ***loop:*** declara la estructura de los bucles “while” y “for” para que se considere correcto.

Manual de Usuario

Instalación:

Descarga del archivo .flex llamado lenguaje.flex, así como el archivo .cup llamado lenguaje.cup y el archivo de prueba.

Integración es el espacio de trabajo Eclipse, como se haría con cualquier archivo .java.

Refrescar el proyecto donde se ha integrado.

Consideraciones:

Previamente se tiene que haber instalado la extensión JFlex y de Cup para su correcto funcionamiento, descargando las extensiones e integrando las en la aplicación usando su respectivo método de instalación.

No requiere de compilación manual, Eclipse la realiza de forma automática.

Ejecución (desde el entorno de Eclipse):

Correr el archivo de lenguaje.flex, hecho esto se abrirá el terminal interno de Eclipse.

En el terminal de Eclipse, se pasa por la línea de comandos la estructura o el archivo que contenga la expresión a analizar.

Al presionar la tecla “Enter”, el programa se ejecutara mostrando por pantalla los elementos del lenguaje que ha detectado

Para ejecutar el archivo de lenguaje.cup, hecho esto se abrirá el terminal interno de Eclipse.

En el terminal de Eclipse, se pasa por la línea de comandos la estructura o el archivo que contenga la expresión a analizar.

Al presionar la tecla “Enter”, el programa se ejecutara mostrando por pantalla los elementos del lenguaje que ha detectado

Consideraciones:

Mostrará un mensaje de error en el caso de que no detecte alguno de los definidos en el lenguaje.