**Laboratorio**

**Analizador Sintactico 2 con JAVACC**

Vamos a realizar un analizador sintáctico para un lenguaje que permite declarar variables booleanas y enteres y realizar operaciones matemáticas entre ellas y que adicionalmente imprima un árbol de derivación.

La gramática para el lenguaje es:

<Programa> ::= <Sentencias> <EOF>

<Sentencias> ::= <SimpleStm>";"[<Sentencias>]

<SimpleStm> ::= <expr> | <declaracion>

<declaracion> ::= int <id> | bool <id>

<expr> ::= <Term> (<AddOp> <Term>)\*

<Term> ::= <Factor> (<MulOp> <Factor>)\*

<Factor> ::= "("<expr>")" | <number> | <identifier>

<MulOp> ::= "\*"|"/"

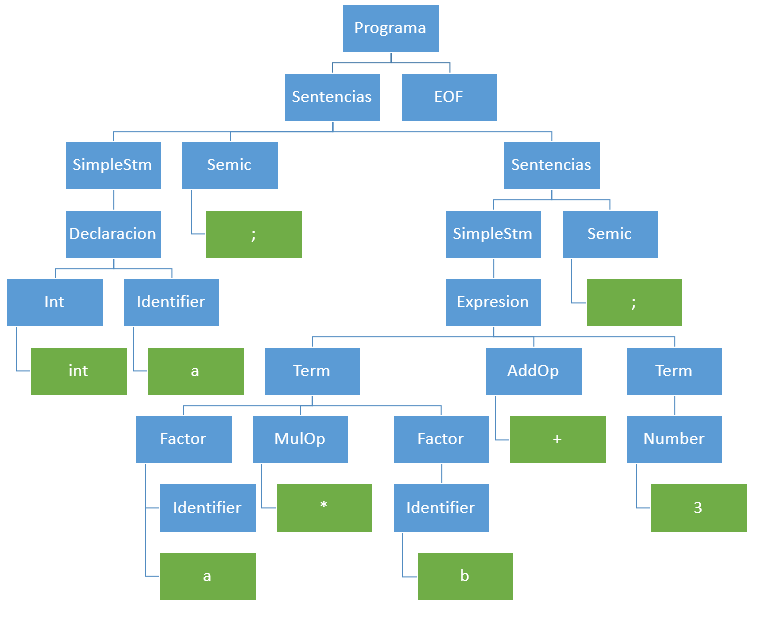
<AddOp> ::= "+"|"-"

Un ejemplo de código con el lenguaje descrito es:

int a;

a\*b+3;

Lo que queremos que aparezca al ejecutar el programa es el árbol de derivación



Como todo lo hacemos por consola, entonces vamos a mostrarlo de forma escrita deberemos obtener algo como esto:

Programa

Sentencias

SimpleStm

Declaracion

* + - * 1. Int (int)
        2. Identifier (a)

Semic (;)

Sentencias

* + - 1. SimpleStm
         1. Expresion

Term

Factor

Identifier (a)

MulOp (\*)

Factor

Identifier (b)

AddOp (+)

Term

Number (3)

* + - 1. Semic (;)

EOF

Una vez que tenemos claro el lenguaje y sobre todo la gramática, podemos desarrollar el analizador semántico en JAVACC muy fácilmente para eso siga los siguientes pasos:

Usando JAVACC en eclipse.

1. Cree un proyecto en JAVA (Java Project)
2. Una vez que el proyecto esta creado, seleccionar la carpeta src y dar clic derecho sobre ella.
3. Seleccione New->File.
4. En la ventana que sale, escriba comp.jjt y luego de clic en el botón finish.
5. Escriban en siguiente código.

/\*\*

\* JavaCC template file created by SF JavaCC plugin 1.5.28+ wizard for JavaCC 1.5.0+

\*/

**options**

**{**

**static** = **true**;

**}**

**PARSER\_BEGIN(comp)**

**public** **class** comp

{

**public** **static** **void** main(String args []) **throws** ParseException

{

**try**

{

System.out.println("Ingrese el código a analizar:");

comp analizador = **new** comp(System.in);

analizador.Programa();

System.out.println("Se ha compilado con éxito");

}

**catch**(ParseException e)

{

System.err.println("Se han encontrado errores");

System.err.println(e.getMessage());

}

}

}

**PARSER\_END(comp)**

**SKIP**: /\* Whitespace \*/

**{**

"\t"

**|** "\n"

**|** "\r"

**|** " "

**}**

**TOKEN**:

**{**

**<**LPAREN: "("**>**

**|** **<**RPAREN: ")"**>**

**|** **<**ADD\_OP: "+" **|** "-"**>**

**|** **<**MULT\_OP: "\*" **|** "/"**>**

**|** **<**INT: "int"**>**

**|** **<**BOOL: "bool"**>**

**|** **<**NUMBER: **([**"0"**-**"9"**])+>**

**|** **<**ID: **([**"a"**-**"z","A"**-**"Z"**])+>**

**|** **<**SEMIC: ";"**>**

**|** **<**FINARCH: "?"**>**

**}**

**void** Programa():

**{}**

**{**

Sentencias() <FINARCH>

**}**

**void** Sentencias(): **{}**

**{**

**(**SimpleStm() <SEMIC> **[**Sentencias()**]** **)**

**}**

**void** SimpleStm(): **{}**

**{**

expresion() **|** declaracion()

**}**

**void** declaracion(): **{}**

**{**

<INT> <ID> **|** <BOOL> <ID>

**}**

**void** expresion(): **{}**

**{**

term() **(**<ADD\_OP> term()**)\***

**}**

**void** term() : **{}**

**{**

factor()**(**<MULT\_OP> factor()**)\***

**}**

**void** factor() : **{}**

**{**

<LPAREN> expresion() <RPAREN>

**|** <NUMBER>

**|** <ID>

**}**

1. Ahora vaya al método main en este mismo archivo y ubiquen la línea

comp analizador = **new** comp(System.in);

Debajo de esa linea, van a declarar una variable de tipo SimpleNode.

1. Van a asignarle a esa variable, el inicio del análisis sintáctico.
2. Posteriormente escriben el nombre de la variable SimpleNode que crearon y llaman la propiedad dump, a la que le pasaran una cadena vacía (Por ejemplo n.dump(“”) )
3. Adicionen otro catch que capture una excepción cualquiera, por si ocurre algo con el árbol.
4. Ahora van a la categoría sintáctica Programa y al final del token <FINARCH> escriben { return jjtThis; } y cambian la declaración de esa categoría para que no sea de tipo void, si no de tipo SimpleNode.
5. Guarden todos los cambios y ahora en el explorador de archivos de eclipse ubique el archivo que crearon (comp.jjt) y le dan clic derecho, buscan la opción Compile with javacc |jjtree | … y dan clic en esa opción.
6. Se generan alrededor de 11 archivos nuevos.
7. Para probarlo vayan a la clase comp.java y dar clic derecho Run as -> Java Aplication

Si ingresamos el código

int a;

a\*b+3;

?

Tendremos la salida en pantalla

Ingrese el código a analizar:

int a;

a\*b+3;

?

Programa

Sentencias

SimpleStm

declaracion

Sentencias

SimpleStm

expresion

term

factor

factor

term

factor

Se ha compilado con éxito

**Pero esto no se parece mucho a lo que queremos que salga.**

Observen cuidadosamente, el solo está imprimiendo las categorías sintácticas, pero no está imprimiendo los tokens, ni los lexemas.

**Para que salgan los tokens**

1. Ubiquen la categoría sintáctica Declaración. Si observan bien, ella esta compuesta de 2 alternativas, declarar un entero o un boleano. Ambas alternativas están formadas de 2 tokens seguidos, estos tokens no se muestran en la salida por consola.
2. Seguido del token <INT>, vamos a escribir #ENTERO(true); esta es la forma en que crearemos nodos propios.
3. Seguido del token <ID>, vamos a escribir #IDEN(true).
4. Ahora guardemos y compilemos de nuevo
5. Luego ejecutemos comp.java e ingresemos el código de ejemplo.

Debieron obtener una salida como esta:

Ingrese el código a analizar:

int a;

a\*b+3;?

Programa

Sentencias

SimpleStm

declaracion

ENTERO

IDENT

Sentencias

SimpleStm

expresion

term

factor

factor

term

factor

Se ha compilado con éxito

Como pueden apreciar, debajo de declaración apareció ENTERO e IDENT. Pero esto aún le falta para parecerse a lo que necesitamos, ya que ahora salen las categorías sintácticas y los tokens, pero no salen los lexemas.

**Para que salgan los lexemas:**

1. De nuevo nos ubicamos en la categoría sintáctica Declaración, ahora buscaremos {} que aparecen después de Declaracion, dentro de los corchetes vamos a declarar una variable de tipo Token, por ejemplo Token t;
2. Luego vamos a asignarle a t el token
3. Vamos a asignarle al valor por defecto del nodo, el lexema asi: jjtThis.value = t.image.
4. Debe quedar algo como esto:

**void** declaracion(): **{** Token t; **}**

**{**

t= <INT> **{**jjtThis.value = t.image;**}** #ENTERO(true) t = <ID> **{**jjtThis.value = t.image;**}** #IDENT(**true**) | <BOOL><ID>

**}**

1. Compilamos todo, y luego abrimos el archivo SimpleNode.java y ubicamos el método dump
2. Buscamos dentro de este método la línea

System.***out***.println(toString(prefix));

1. Cambiamos que no sea println, si no solo print y debajo de este vamos a poner a imprimir el value que asignamos en el paso 4 así:

**if** (value != **null**)

{

System.***out***.print(" (" + value + ")");

}

System.***out***.println();

1. Grabamos y ejecutamos el archivo comp.java y volvemos a pasar el código de prueba

Deben obtener una salida como esta:

Ingrese el código a analizar:

int a;

a\*b+3;?

Programa

Sentencias

SimpleStm

declaracion

ENTERO (int)

IDENT (a)

Sentencias

SimpleStm

expresion

term

factor

factor

term

factor

Se ha compilado con éxito

Ahora si se parece más a lo que queremos

Para terminar, ustedes deben completar el programa (comp.jjt), para que salgan completos las categorías gramaticales, los tokens y los lexemas.

Un ejemplo de una salida cuando este terminado.

int a;

bool b;

a\*(b+x)/3;?

Programa

Sentencias

SimpleStm

declaracion

ENTERO (int)

IDENT (a)

PUNTOYCOMA (;)

Sentencias

SimpleStm

declaracion

BOLEANO (bool)

IDENT (b)

PUNTOYCOMA (;)

Sentencias

SimpleStm

expresion

term

factor

IDEN (a)

OPERMUL (\*)

factor

ABRIRP (()

expresion

term

factor

IDEN (b)

OPERADD (+)

term

factor

IDEN (x)

CERRARP ())

OPERMUL (/)

factor

NUMERO (3)

PUNTOYCOMA (;)

Se ha compilado con éxito

**Al terminar deben subir el archivo comp.jjt y el archivo SimpleNode.java en el moodle, en la tarea del laboratorio.**