UNIVERSIDAD NACIONAL DE LOJA

AREA DE LA ENERGIA, LAS INDUSTRIAS Y LOS RECURSOS NATURALES NO RENOVABLES

INGENIERIA EN SISTEMAS

Desarrollo de un Compilador en español para la ejecución de algoritmos en pseudocódigo

Tesis Previa a la Obtención del Título de Ingeniero en Sistemas

Autores:

Alondra María Ordóñez Ordóñez

Alex Patricio Román Macas

Director:

Ing. Edison Leonardo Coronel Romero

Loja-Ecuador

2011

CERTIFICACIÓN DE DIRECTOR

Loja, Noviembre de 2011

Ing. Edison Leonardo Coronel Romero
DIRECTOR DE TESIS

CERTIFICA:

Que el Sr. Alex Patricio Román Macas y la Srta. Alondra María Ordóñez Ordóñez, autores de la Tesis: **DESARROLLO DE UN COMPILADOR EN ESPAÑOL PARA LA EJECUCIÓN DE ALGORITMOS EN PSEUDOCODIGO**, han cumplido con todos los requerimientos y requisitos que contempla el reglamento general de la Universidad Nacional de Loja, además todo el proceso de desarrollo fue coordinado y revisado por mi persona, por lo que autorizo su presentación y sustentación.

Es todo cuanto puedo certificar honor a la verdad.

Ing. Edison Leonardo Coronel Romero
DIRECTOR DE TESIS

AUTORÍA

Las ideas, opiniones, comentarios, conclusiones y recomendaciones que se encuentran en el presente proyecto son absoluta responsabilidad de los autores.

Alondra María Ordóñez Ordóñez

Alex Patricio Román Macas

DEDICATORIA

A Dios por iluminar mi camino y mostrarme lo bella que es la vida.

A mis padres por el apoyo brindado a lo largo de mi formación, que me permiten crecer día a día.

A mis hermanos porque gracias a su amistad incondicional, han contribuido para alcanzar mis metas.

A mi novia por su incondicional apoyo en momentos difíciles de mi camino que permitieron ser fuerte ante las adversidades.

Alex

A Dios porque es mi guía, mi luz y mi inspiración en cada momento de mi vida y porque nunca me ha dejado sola.

A mi padre que desde el cielo ha estado siempre conmigo dándome su bendición.

A mi madre porque gracias a ella y a sus esfuerzos he podido lograr mis metas y por que con su ejemplo me ha enseñado a ser una persona de bien.

A mis hermanas y mi sobrina porque ellas son mi compañía y apoyo, porque su amor y el de mi madre me dan fuerzas para salir adelante.

Alondra

AGRADECIMIENTO

En primer lugar a Dios ya que gracias a Él hemos podido llegar hasta este punto, por todas la bendiciones que nos da día a día y por iluminar nuestras vidas de manera tan generosa.

A la Ing. Mireya Erreyes por su colaboración durante las pruebas de validación del software y por sus consejos que nos sirvieron de mucha ayuda.

Al Ing. Edison Coronel, director de tesis, quien con su asesoramiento, conocimiento y experiencia nos ayudó a culminar con éxito nuestro proyecto.

A todos los familiares y amigos que han estado cerca, con sus buenos deseos y dándonos ánimos de seguir adelante, ya que también han sido un gran apoyo y un incentivo para llegar a nuestra meta.

Alondra María
Alex Patricio

CESIÓN DE DERECHOS

Alondra María Ordóñez Ordoñez y Alex Patricio Román Macas, autores del presente Trabajo de Tesis certifican la propiedad intelectual a la Universidad Nacional de Loja, y autorizan a la misma para hacer uso del presente documento como considere conveniente.

Alondra María Ordóñez Ordóñez

Alex Patricio Román Macas

A.TITULO

Desarrollo de un Compilador en español para la ejecución de algoritmos en pseudocódigo.

B.RESUMEN

El presente documento contiene todo lo referente al desarrollo de un compilador en español o Software Base para la unidad de Metodología de la programación de la carrera de Ingeniería en Sistemas de la Universidad Nacional de Loja.

El compilador llamado ARK es un programa que lee otro programa escrito en un pseudocódigo (lenguaje fuente), y lo traduce a un programa equivalente en lenguaje JAVA (el lenguaje objeto). Como parte importante de este proceso de traducción, el compilador ARK informa al usuario de la presencia de errores en el programa fuente en este caso el pseudocódigo en español.

Para el desarrollo del compilador ARK se utilizó el framework ANTLR (ANother Tool for Languaje Recognition, Otra Herramienta Para Reconocimiento de Lenguajes), una herramienta que opera sobre lenguajes, proporcionando un marco para construir reconocedores, interpretes, compiladores y traductores de lenguajes a partir de las descripciones gramaticales de los mismos.

El compilador ARK es una herramienta que servirá principalmente a los alumnos de cuarto y quinto módulo de la carrera de Ingeniería en Sistemas, así como a otras personas interesadas en la implementación de algoritmos en pseudocódigo.

SUMMARY

This document contains everything related to the development of a compiler in Spanish or Base Software for the unity of Programming Methodology of Systems Engineering in National University of Loja.

The compiler called ARK is another program that reads a program written in a pseudo-language (font language), and translates it into an equivalent program in Java language (the object language). As part of this process of translation, the compiler informs the user ARK for errors in the source in this case the pseudocode in Spanish.

For the development of ARK compiler was used framework ANTLR (Another Tool for Language Recognition, Another tool for language recognition), a tool that operates on languages, providing a framework for constructing recognizers, interpreters, compilers and translators of languages from grammatical descriptions of the same.

The compiler ARK is a tool that will primarily serve students in fourth and fifth module of Systems Engineering, as well as others interested in the implementation of algorithms in pseudocode.

ÍNDICE

INDICE DE FIGURAS

INDICE DE TABLAS

C.INTRODUCCIÓN

La Universidad Nacional de Loja tiene como misión formar profesionales capaces de solucionar los problemas que se presenten en la sociedad; para ello pretende acercar el trabajo de los estudiantes a la comunidad de la que son parte y fomentar en ellos el espíritu de colaboración y un sentido más humano de la sociedad que los rodea.

El Área de la Energía, las Industrias y los Recursos Naturales no Renovables, toma esta perspectiva desde el punto de vista técnico que implementa la investigación de campo para crear profesionales que resuelvan dificultades con métodos adecuados y acordes a las necesidades de la sociedad actual.

La Carrera de Ingeniería en Sistemas, promueve la solución de los problemas de la vida diaria mediante un enfoque diferente, utilizando los diversos avances tecnológicos con los que contamos hoy en día.

Entre los avances tecnológicos mencionados, se encuentran los compiladores, un compilador es un programa que lee un archivo escrito en un lenguaje fuente y lo traduce a un programa equivalente al lenguaje objeto, como parte del proceso de traducción, el compilador informa al usuario de la existencia de errores en el programa fuente. Este tipo de programas son de mucha ayuda en la carrera de Ingeniería en Sistemas sobre todo en las unidades de Introducción a la Programación, las mismas que se imparten a partir del cuarto modulo en la carrera.

Por lo mencionado anteriormente, y haciendo uso del Sistema Académico Modular por Objetos de Transformación, el cual conlleva al desarrollo de un proceso investigativo, se ha creído conveniente enfocar la investigación a la Unidad de Metodología de la Programación que se imparte en el Cuarto Módulo de la carrera de Ingeniería en Sistemas de la Universidad Nacional de Loja, para lo cual se plantea como objetivo principal el desarrollo de un compilador en español para la ejecución de algoritmos en pseudocódigo, el cumplimiento de este objetivo conducirá a demostrar la hipótesis que

plantea que el uso de esta herramienta ayudará a mejorar y agilizar el proceso de enseñanza-aprendizaje de los alumnos de Cuarto y Quinto Módulo de la Carrera de Ingeniería en Sistemas de la Universidad Nacional de Loja.

El presente proyecto constituye un aporte importante ya que está enfocado a la resolución de problemas en un menor tiempo, mejorando la situación actual, además no se han realizado proyectos de esta naturaleza anteriormente por lo que también representa un aporte original e interesante para el Área y la Universidad.

Es muy importante la colaboración de todas las personas relacionadas con el proyecto para que los resultados obtenidos sean los esperados; en primer lugar están los alumnos y los docentes de Cuartos y Quintos Módulos de la carrera de Ingeniería en Sistemas quienes conocen más de cerca los problemas que existen por la falta de una herramienta de apoyo, la manera de llegar a ellos es a través de una encuesta estructurada en la que se mencionarán con exactitud dichos problemas a los cuales se pretende dar solución mediante la investigación.

En la metodología utilizada en el presente proyecto, primeramente aplicamos el método deductivo e inductivo para determinar el problema existente y las causas que lo originan para poder plantear soluciones, para lo cual se utilizó la entrevista estructurada con los estudiantes de quinto módulo quienes serán los usuarios de la aplicación.

En cuanto a lo referente a la metodología para el desarrollo del software se utilizó una combinación entre SCRUM y XP, por lo que el presente informe incluye la información que se creyó necesaria para llevar a cabo el proyecto con estas metodologías, como Historia de Usuario, tarjetas CRC, Requerimientos, Glosario de Términos y Modelo de Dominio.

En el Marco teórico, es una base científica en donde se definen contenidos relevantes para el desarrollo del Compilador en el cual se detalla las etapas o fases que lo conforman, en primer lugar está el análisis léxico, en el cual se definen los lexemas y tokens así como los Autómatas Finitos Deterministas, seguidamente se encuentra el

análisis sintáctico, para el que necesitamos conocer las gramáticas EBNF y el cálculo de primeros y siguientes, luego se detalla el análisis semántico con los AST, y la generación de código; luego de detallar las fases del compilador se incluye una breve guía sobre el uso del framework ANTLR (ANother Tool for Language Recognition; en español "otra herramienta para reconocimiento de lenguajes"), una herramienta que opera sobre lenguajes, proporcionando un marco para construir reconocedores (parsers), intérpretes, compiladores y traductores de lenguajes a partir de las descripciones gramaticales de los mismos, permitiendo la identificación de errores léxicos, sintácticos y semánticos[1], se utilizó esta herramienta para el desarrollo de la aplicación.

En los resultados del presente proyecto de tesis, consta el detalle de cada una de las fases del compilador y su implementación, como son el análisis léxico, análisis sintáctico, análisis semántico y generación de código.

Una vez lograda la investigación, como resultado se obtiene un compilador en español para la ejecución de algoritmos en pseudocódigo, compuesto por Analizador Léxico, Analizador Sintáctico, Analizador Semántico y Generador de Código Intermedio, con el cual se realizan las respectivas pruebas de validación que demuestren que se encuentra apto para satisfacer las necesidades de los usuarios.

1 ANother Tool for Language Recognition, 2010. **URL:** http://es.wikipedia.org/wiki/ANTLR

D.MARCO TEÓRICO

CAPITULO I

1. Compiladores

1.1. Análisis Léxico

Analizador léxico (scanner): lee la secuencia de caracteres del programa fuente, carácter a carácter, y los agrupa para formar unidades con significado propio, *los componentes léxicos* (*tokens* en ingles)[2]. Estos componentes léxicos representan:

palabras reservadas: if, while, do, . . .
identificadores: asociados a variables, nombres de funciones, tipos definidos por el usuario, etiquetas,... Por ejemplo: posicion, velocidad, tiempo, . . .
operadores: = * + - / == > < & ! = . . .
símbolos especiales: ; ( ) [ ] *f g* ...
constantes numéricas: literales que representan valores enteros, en coma flotante, etc, 982, 0xF678, -83.2E+2,...
constantes de caracteres: literales que representan cadenas concretas de caracteres, \hola mundo",...

El analizador léxico opera bajo petición del analizador sintáctico devolviendo un componente léxico conforme el analizador sintáctico lo va necesitando para avanzar en la gramática. Los componentes léxicos son los símbolos terminales de la gramática.

[2] **Morales Luna Guillermo,** Árboles de Derivación, 2000 **URL:** http://delta.cs.cinvestav.mx/~gmorales/ta/node104.html

Suele implementarse como una subrutina del analizador sintáctico. Cuando recibe la orden obtén el siguiente componente léxico, el analizador léxico lee los caracteres de entrada hasta identificar el siguiente componente léxico.

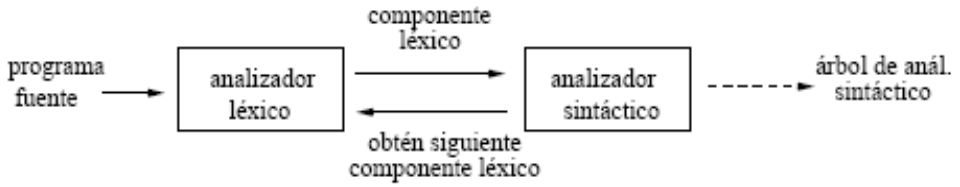


Figura 1. Análisis Léxico

Patrón: es una regla que genera la secuencia de caracteres que puede representar a un determinado componente léxico (una expresión regular).

Lexema: cadena de caracteres que concuerda con un patrón que describe un componente léxico.

Componente léxico: puede tener uno o infinitos lexemas. Por ejemplo: palabras reservadas tienen un único lexema. Los números y los identificadores tienen infinitos lexemas.

Los componentes léxicos se suelen definir como un tipo enumerado. Se codifican como enteros. También se suele almacenar la cadena de caracteres que se acaba de reconocer (el lexema), que se usaría posteriormente para el análisis semántico.[3]

1.1.1. Autómata Finito Determinista (AFD)

Un AFD o autómata finito determinista es aquel en el cual, para cada par (estado, símbolo) está perfectamente definido el siguiente estado al cual pasará el autómata, es decir para cualquier estado q en que se encuentre el autómata y con cualquier símbolo s del alfabeto leído, existe exactamente una transición que parte de q y está etiquetada con s.

[3] Louden, K.C, Construcción de Compiladores*: Principios y Práctica*, 1997, Tema 2, páginas: 31-93.

Formalmente, un autómata finito determinista (AFD) es similar a un Autómata de estados finitos, representado con una 5-tupla (S,Σ,T,s,A) donde:

- Σ es un alfabeto;
- S un conjunto de estados;
- T es la función de transición: $T: S \times \sum \rightarrow S$
- $s \in S$ es el estado inicial;
- $A \subseteq S$ es un conjunto de estados de aceptación o finales.

Al contrario de la definición de autómata finito, este es un caso particular donde no se permiten transiciones vacías, el dominio de la función T es S (con lo cual no se permiten transiciones desde un estado de un mismo símbolo a varios estados).

A partir de este autómata finito es posible hallar la expresión regular resolviendo un sistema de ecuaciones.

$$S_1 = 1\ S_1 + 0\ S_2 + \varepsilon$$

$$S_2 = 1\ S_2 + 0\ S_1$$

Siendo ε la palabra nula. Resolviendo el sistema y haciendo uso de las reducciones apropiadas se obtiene la siguiente expresión regular: 1*(01*01*)*.

1.1.1.1. Transiciones

En Teoría de autómatas y Lógica secuencial, una **tabla de transición de estados** es una tabla que muestra que estado (o estados en el caso de un Autómata finito no determinista) se moverá la máquina de estados, basándose en el estado actual y otras entradas. Una *tabla de estados* es esencialmente una tabla de verdad en la cual algunas de las entradas son el estado actual, y las salidas incluyen el siguiente estado, junto con otras salidas.

1.1.1.1.1. Tablas de Estados de dos dimensiones

Las tablas de transición de estados son normalmente tablas de dos dimensiones. Hay dos formas comunes para construirlas.

- La dimensión vertical indica los Estados Actuales, la dimensión horizontal indica eventos, y las celdas (intersecciones fila/columna) de la tabla contienen el siguiente estado si ocurre un evento (y posiblemente la acción enlazada a esta transición de estados)[4].

Tabla 1. Ejemplo de Tabla de Transición de Estados

Tabla de Transición de Estados				
Events State	E1	E2	...	En
S1	-	Ay/Sj	...	-
S2	-	-	...	Ax/Si
...	...	...	...	...
Sm	Az/Sk	-	...	-

[4] **Sánchez Isabel, Cárdenas María Antonia**, Teoría de autómatas y lenguajes formales, **URL:** http://www.yakiboo.net/apuntes/TALF%20-%20YakiBoo.net.pdf**.**

Ejemplo:

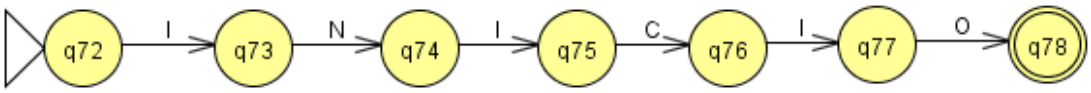


Figura 2. Ejemplo de Autómata Finito Determinista

a. Definición Formal:

- Σ={I, N, I, C, I, O}
- *S*= {q72, q73, q74, q75, q76, q77, q78}
- *T*= {q72 → q73

  q73 → q74

  q74 → q75

  q75 → q76

  q76 → q77

  q77 → q78}
- S= {q72}
- A= {q78}

b. Tabla de Transición de Estados

Tabla 2. Tabla de transición de AFD de INICIO

Estados	q72	q73	q74	q75	q76	q77	q78
q72	-	I	-	-	-	-	-
q73	-	-	N	-	-	-	-
q74	-	-	-	I	-	-	-
q75	-	-	-	-	C	-	-
q76	-	-	-	-	-	I	-
q77	-	-	-	-	-	-	O
q78	-	-	-	-	-	-	-

1.2. Análisis Sintáctico

Un analizador sintáctico (en inglés parser) es una de las partes de un compilador que transforma su entrada en un árbol de derivación.
El análisis sintáctico convierte el texto de entrada en otras estructuras (comúnmente árboles), que son más útiles para el posterior análisis y capturan la jerarquía implícita de la entrada. Un analizador léxico crea tokens de una secuencia de caracteres de entrada y son estos tokens los que son procesados por el analizador sintáctico para construir la estructura de datos, por ejemplo un árbol de análisis o árboles de sintaxis abstracta.

El análisis sintáctico también es un estado inicial del análisis de frases de lenguaje natural. Es usado para generar diagramas de lenguajes que usan flexión gramatical, como los idiomas romances o el latín. Los lenguajes habitualmente reconocidos por los analizadores sintácticos son los lenguajes libres de contexto. Cabe notar que existe una justificación formal que establece que los lenguajes libres de contexto son aquellos reconocibles por un autómata de pila, de modo que todo analizador sintáctico que reconozca un lenguaje libre de contexto es equivalente en capacidad computacional a un autómata de pila.

1.2.1. Gramáticas LL(k)

Las gramáticas LL(k) son un subconjunto de las gramáticas libres de contexto.
Permiten un análisis descendente determinista (o sin retroceso), por medio del reconocimiento de la cadena de entrada de izquierda a derecha ("*Left to righ*t") y que va tomando las derivaciones más hacia la izquierda ("*Leftmos*t") con sólo mirar los *k* tokens situados a continuación de donde se halla. Si k=1 se habla de gramáticas LL(1).
Las gramáticas LL(1) permiten construir un analizador determinista descendente con tan sólo examinar en cada momento el símbolo actual de la cadena de entrada (símbolo de pre análisis) para saber qué producción aplicar.

1.2.1.1. Teorema

- Una gramática LL(k) es no ambigua.
- Una gramática LL(k) no es recursiva a izquierdas[5].

1.2.1.2. Gramáticas BNF

La definición formal de la sintaxis de un Lenguaje de Programación se conoce comúnmente como **gramática**. Una gramática se compone de un conjunto de reglas (producciones) que definen las palabras (elementos léxicos) y unidades sintácticas. Una gramática formal es una que usa una notación definida (o metalenguaje) de manera estricta.
Cuando se considera la estructura de una oración en español, se le describe por lo general como una secuencia de categorías:

sujeto | verbo | complemento

Podemos decir que una oración puede ser una oración declarativa o una oración interrogativa, lo que denotamos:

<oración> ::= <declarativa> | <interrogativa>

donde "::=" significa "se define cómo" y "|" significa "o"; comparándolo con el lenguaje usado en las gramáticas formales, las palabras encerradas entre <palabra> son categorías sintácticas o no terminales, es decir deben ser definidas; "::=" equivale a "->" y "|" tiene el mismo significado. Así, por ejemplo, en gramáticas regulares:

5 LUENGO, Cándida, Análisis Sintáctico en Procesadores de Lenguaje, Oviedo, Enero 2005.

O -> D | I

S -> S V C

S -> A N

I -> ¿ V S P ?

en nuestro metalenguaje:

<oración> ::= <declarativa> | <interrogativa>

<declarativa> ::= <sujeto> <verbo> <complemento>

<sujeto> ::= <artículo> <nombre>

:

<interrogativa> ::= ¿ <verbo> <sujeto> <predicado> ?

:

Esta notación específica se conoce como BNF (Backus Naur Form) desarrollada por John Backus alrededor de 1960 para la definición de ALGOL.

Para expresar repetición se usa la recursividad, por ejemplo:

<entero> ::= <entero> <dígito> | <dígito>

define a un entero como una secuencia de dígitos, con al menos un dígito.

A pesar de su estructura sencilla, BNF sirve para definir casi todos los lenguajes de programación. Las áreas de sintaxis que no pueden definirse con una BNF y tampoco con una gramática libre del contexto, son aquellas que indican dependencia contextual. Por ejemplo, "un arreglo declarado con dos dimensiones, no se puede referenciar con tres subíndices".

1.2.1.2.1. Gramáticas EBNF

Extended Backus–Naur Form utiliza las extensiones siguientes que permiten realizar descripciones más fáciles de los lenguajes:

Tabla 3. Simbología de Gramáticas EBNF

Sintaxis	Significado
::=	se define como
t'	el símbolo terminal t
<nt>	el símbolo no terminal nt
(...)	usado para agrupar
*	cero o más repeticiones del elemento anterior
+	una o más repeticiones del elemento anterior
[...]	elemento discrecional
|	alternativa de varias formas sintácticas válidas

Ejemplo:

<entero> ::= [+|-] <dígito> {<dígito>}*
<identificador> ::= <letra> {<letra> | <dígito>}*

1.2.1.3. Árbol de sintaxis abstracta (AST)

Los ASTs (*Abstract Syntax Trees*, o Árboles de Sintaxis Abstracta) sirven para manejar la información semántica de un código. La forma más eficiente de manejar la información proveniente de un lenguaje de programación es la forma arbórea; por eso la estructura de datos elegida es un árbol. Además, construyendo ASTs a partir de un texto podemos obviar mucha información irrelevante; si un AST se construye bien, no habrá que tratar con símbolos de puntuación o azúcar sintáctica en el nivel semántico.

Al contrario que los flujos, una estructura en árbol puede especificar la relación jerárquica entre los símbolos de una gramática.

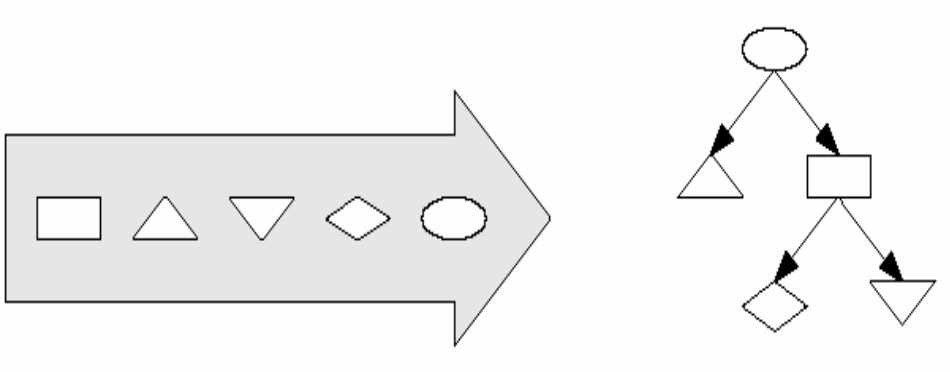


Figura 3. Árbol de sintaxis abstracta (AST)

Los ASTs pueden intervenir en varias fases del análisis: como producto del análisis sintáctico, como elemento intermedio en sucesivos análisis semánticos y como entrada para la generación de código[6].

1.3. Análisis Semántico y Código Intermedio

Se compone de un conjunto de rutinas independientes, llamadas por los analizadores morfológico y sintáctico.

El análisis semántico utiliza como entrada el árbol sintáctico detectado por el análisis sintáctico para comprobar restricciones de tipo y otras limitaciones semánticas y preparar la generación de código.

En compiladores de un solo paso, las llamadas a las rutinas semánticas se realizan directamente desde el analizador sintáctico y son dichas rutinas las que llaman al generador de código. El instrumento más utilizado para conseguirlo es la gramática de atributos.

[6]Árbol de sintaxis Abstracta, URL: [http://www.uco.es/~ma1fegan/Comunes/manuales/pl/ANTLR/Arboles-de-sintaxis-abstracta.pdf]

En compiladores de dos o más pasos, el análisis semántico se realiza independientemente de la generación de código, pasándose información a través de un archivo intermedio, que normalmente contiene información sobre el árbol sintáctico en forma linealizada (para facilitar su manejo y hacer posible su almacenamiento en memoria auxiliar).

En cualquier caso, las rutinas semánticas suelen hacer uso de una pila (la pila semántica) que contiene la información semántica asociada a los operandos (y a veces a los operadores) en forma de *registros semánticos*.

1.4. Generación De Código

El analizador sintáctico va generando acciones que valida el analizador semántico y que se convierten en tercetos. Esta conversión en tercetos constituye el generador de código intermedio.

Dado que el lenguaje puede presentar distintas funciones anidadas, los tercetos los generamos por orden del parser y son almacenados en un sitio u otro dependiendo del contexto en que nos encontremos. Es decir, se almacenan en una lista de tercetos dependiente de la Tabla de Símbolos. Hay tantas listas de tercetos como funciones haya en el código fuente más una lista de tercetos asociada a la Tabla de Símbolos Global.

No obstante una vez finalizado el análisis, todos estos tercetos repartidos en distintas listas se vuelcan a una sola lista de tercetos global. Esta será la que finalmente se optimice y a partir de la que se generará el programa en ensamblador.

El problema de tener que manejar tercetos indirectos fue resuelto modificando el método de inserción sobre la lista de tercetos utilizada en cada momento, de manera que se realiza previamente una búsqueda de algún terceto que sea exactamente igual al que estamos insertando. En caso afirmativo, insertamos en la lista no un terceto nuevo, sino un puntero al ya existente, y marcamos dicho terceto como terceto indirecto. Son tercetos indirectos aquellos **marcados con un asterisco** después del índice en los volcados de la lista de tercetos.

1.4.1. Optimización y Generación De Código Final

En la etapa de optimización de código, se busca obtener el código más corto y rápido posible, utilizando distintos algoritmos de optimización.

Etapa de generación de código**,** final se lleva el código intermedio final a código maquina o código objeto, que por lo general consiste en un código maquina relocalizable o código ensamblador. Se selecciona las posiciones de memoria para los datos (variables) y se traduce cada una de las instrucciones intermedias a una secuencia de instrucciones de maquina puro. [7]

[7] Louden, K.C, Construcción de Compiladores*: Principios y Práctica*, 1997, Tema 2.

CAPITULO II

2. ANTLR (ANOTHER TOOL FOR LANGUAGE RECOGNITION)

La mejor manera de aprender acerca de ANTLR es caminar a través de un ejemplo simple pero útil. En este capítulo, se va a construir un evaluador de expresiones aritméticas que soporta un número reducido de operadores y asignaciones de variables. De hecho, se va a implementar el evaluador de dos maneras diferentes. En primer lugar, se va a construir una gramática de análisis sintáctico para reconocer las expresiones del lenguaje y luego agregar acciones para evaluar la realidad e imprimir el resultado. En segundo lugar, se va a modificar la gramática del analizador sintáctico para generar una forma intermedia de árboles de estructura de datos en lugar calcular inmediatamente los resultados. A continuación, creará un árbol gramatical para recorrer estos los árboles, la adición de acciones destinadas a evaluar e imprimir el resultado. Una vez terminado esto, se tendrá una buena visión general de cómo construir traductores con ANTLR. Se aprenderá acerca del analizador sintáctico de gramáticas, símbolos, acciones, AST, y árboles gramaticales por ejemplo.

Para hacerlo simple, se va a restringir el lenguaje de expresiones de apoyo a las siguientes construcciones:

• Los operadores más, menos, y multiplicación con el orden habitual de la evaluación del operador, lo que permite expresiones como ésta:

3 +4 * 5-1

• Las expresiones entre paréntesis alteran el orden de evaluación del operador, lo que permite expresiones como ésta:

(3 +4) * 5

• Las asignaciones de variables y referencias, lo que permite expresiones como éstas:

a = 3

```
b = 4
2 + a * b
```

Esto es lo se quiere que el traductor haga: cuando se vea 3 +4, debe emitir 7. Cuando ve dogs = 21, se debe asignar a los dogs valor 21. Si el traductor nunca ve los dogs una vez más, se debe pretender que escribe 21 en lugar de dogs. ¿Cómo podemos siquiera comenzar a resolver este problema? Bueno, hay dos tareas generales:

- Construir una gramática que describe la estructura general de las expresiones sintácticas y asignaciones. El resultado de ese esfuerzo es un reconocedor que responde sí o no, si la entrada era una expresión o una asignación válida.
- Insertar código de inserción entre los elementos de la gramática en las posiciones adecuadas para evaluar las partes de la expresión. Por ejemplo, dada la entrada 3, el traductor debe ejecutar una acción que convierte el carácter a su valor entero. Para la entrada de 3 +4, el traductor debe ejecutar una acción que agrega los resultados de dos ejecuciones acción anterior, a saber, las acciones que convierten los caracteres 3 y 4 a sus equivalentes enteros.

Después de completar las dos grandes tareas, tendremos un traductor que traduce expresiones para el valor de la aritmética usual. Vamos a seguir este proceso:

1. Construir la gramática de la expresión.
2. Examinar los archivos generados por ANTLR.
3. Construir una instalación de prueba y prueba del reconocedor.
4. Agregar acciones a la gramática para evaluar expresiones y emitir los resultados.
5. Aumentar el banco de pruebas y probar el traductor.

Ahora que se ha definido el lenguaje, vamos a construir una gramática de ANTLR que reconoce sentencias en el lenguaje y calcula los resultados.

2.1. Reconociendo la Sintaxis del Lenguaje

Se tiene que construir una gramática que describe completamente la sintaxis de nuestro lenguaje de expresión, incluida la forma de los identificadores y números enteros. De la gramática, ANTLR va a generar un programa que reconoce expresiones válidas, automáticamente con la emisión de errores de las expresiones inválidas. Comience por pensar en la estructura general de la entrada, y luego descomponer esa estructura hacia abajo en subestructuras, y así sucesivamente, hasta llegar a una estructura que no se puede descomponer más. En este caso, la entrada general consiste en una serie de expresiones y asignaciones, lo que vamos a dividirlo en más detalle a medida que avancemos.
La gramática ANTLR más común es una combinación de gramática que especifica tanto el analizador y las reglas léxicas. Estas reglas especifican la estructura gramatical de una expresión, así como su estructura léxica (llamada Token). Por ejemplo, una asignación es un identificador, seguido por un signo igual, seguido por una expresión, y termina con un salto de línea, un identificador es una secuencia de letras. Definir una gramática combinado dándole un nombre con la palabra clave gramática:

```
grammar Expr;
«rules»
```

Se pone esta gramática en Expr.g archivo porque el nombre del archivo debe coincidir con el nombre de la gramática.
Un programa en este lenguaje se parece a una serie de declaraciones seguido por el carácter de nueva línea (nueva línea por sí mismo es una declaración vacía y se omite). Más formalmente, estas reglas de Inglés son similares al siguiente cuando se escribe en notación ANTLR donde: comienzan una definición de la regla y | separa las alternativas de la regla.

```
prog: stat+ ;
stat: expr NEWLINE
```

```
    | ID '=' expr NEWLINE
    | NEWLINE
    ;
```

Una regla gramatical es una lista con nombre de una o más alternativas, tales como prog y stat. Prog se puede leer de la siguiente manera: un prog es una lista de reglas stat. La regla stat se lee de la siguiente manera: un stat es una de las tres alternativas:

- Un expr seguido por un salto de línea (token de línea nueva)
- La secuencia de ID (identificador), '=', expr, NEWLINE
- Un token de línea nueva

Ahora se tiene que definir lo que es una expresión, la regla expr, parece. Resulta que hay un patrón de diseño de la gramática para expresiones aritméticas. El patrón prescribe una serie de reglas, una para cada nivel de prioridad de los operadores y otra para los átomos de más bajo nivel de expresión que describe como enteros.

Comenzar con una regla general llamada expr que representa una expresión completa. Regla expr coinciden los operadores con los más débiles de la precedencia, más y menos, y se referirá a una regla que coincide con subexpresiones para los operadores con mayor prioridad de los siguientes. En este caso, el siguiente operador es multiplicación. Podemos llamar a la regla multExpr. Reglas expr, multExpr, y atom aparece así:

```
expr: multExpr (('+' |'-' ) multExpr)*
    ;
multExpr
    : atom ('*' atom)*
    ;
atom: INT
    | ID
    | '(' expr ')'
```

En cuanto al nivel léxico, vamos a definir los símbolos del vocabulario (tokens): identificadores, números enteros, y el carácter de nueva línea. Cualquier otro espacio en blanco se ignora. Reglas léxicas comienzan con una letra mayúscula en ANTLR y por lo general se refieren a caracteres literales y cadenas, no tokens, no como analizador de reglas. Aquí están todas las reglas léxicas que necesitaremos:

```
ID : ('a'..'z' |'A'..'Z' )+ ;
INT : '0'..'9' + ;
NEWLINE:'\r' ? '\n' ;
WS : (' ' |'\t' |'\n' |'\r' )+ {skip();} ;
```

Regla WS (espacio en blanco) es el único con una acción (skip ();) que le dice a ANTLR que debe saltar lo que acaba de encontrar y buscar otro símbolo. La forma más fácil de trabajar con gramáticas ANTLR es utilizar ANTLRWorks, que proporciona un entorno de desarrollo sofisticado. Tenga en cuenta que la vista del diagrama de sintaxis de una regla hace que sea fácil de entender tal cual es la regla encontrada. En este momento, no tenemos el código de Java para ejecutar. Todo lo que tenemos es una gramática de ANTLR. Para convertir la gramática ANTLR a Java, ANTLR invoca desde la línea de comando (primero debemos asegurarnos de que antlr-3.0.jar, antlr-2.7.7, y StringTemplate 3.0.jar están en el CLASSPATH):

```
$ java org.antlr.Tool Expr.g
ANTLR Parser Generator Version 3.0 1989-2007
$
```

También se puede utilizar para generar código ANTLRWorks con la opción en el menú Generar de generar el código, lo que generará un código en el mismo directorio que el archivo de la gramática.

2.2. ¿Qué Genera ANTLR?

A partir de una gramática combinada, ANTLR genera un analizador y lexer (escrito en Java, en este caso) que se puede compilar. Mejor aún, el código de ANTLR genera es humanamente legible. Se recomienda mirar el código generado, ya que realmente es muy útil; ANTLR genera los siguientes archivos:

Tabla 4. Archivos generados por ANTLR

Archivo Generado	Descripción
ExprParser.java	El analizador sintáctico descendente recursivo generado a partir de la gramática. De la gramática Expr, ANTLR genera ExprParser y ExprLexer.
Expr.tokens	La lista de nombres del token, tipo de asignaciones del token como INT = 6.
ExprLexer.java	El analizador léxico recursivo descendente generada a partir de Expr__.g

Si se observa en el interior ExprParser.java, por ejemplo, podrás ver un método para cada regla definido en la gramática. El código para la regla multExpr se parece al siguiente pseudocódigo:

```
void multExpr() {
        try {
                atom();
                while ( «next input symbol is * » ) {
                        match('*' );
                        atom();
                }
        }
        catch (RecognitionException re) {
                reportError(re); // automatic error reporting and recovery
                recover(input,re);
        }
}
```

El pseudocódigo para la regla atom es la siguiente:

```
void atom() {
        try {
        // predict which alternative will succeed
        // by looking at next (lookahead) symbol: input.LA(1)
                int alt=3;
                switch ( «next input symbol » ) {
                        case INT: alt=1; break;
                        case ID: alt=2; break;
                        case '(' : alt=3; break;
                        default: «throw NoViableAltException»
        }
        // now we know which alt will succeed, jump to it
        switch (alt) {
                case 1 : match(INT); break;
                case 2 : match(ID); break;
                case 3 :
                        match('(' );
                        expr(); // invoke rule expr
                        match(')' );
                        break;
                }
        }
        catch (RecognitionException re) {
                reportError(re); // automatic error reporting and recovery
                recover(input,re);
        }
}
```

Se debe tener en cuenta que las referencias de regla, las llamadas a métodos, y las referencias de token se convierten en llamadas a match(token).

Todas las variables y FOLLOW_multExpr_in_expr160 PushFollow () Las referencias a métodos son parte de la estrategia de recuperación de errores, que siempre quiere saber qué token, podría ser el próximo. En el caso de una muestra faltante o adicional, el reconocedor volverá a sincronizar por saltarse los tokens hasta que se ve una señal en el propio "siguiente" set.

ANTLR genera un archivo que contiene los tipos de tokens en caso de que otra gramática quiera usar las mismas definiciones de tipo simbólico. Tipos de tokens son enteros que representan la "especie" de muestra, al igual que los valores ASCII representan los caracteres:

```
INT=6
WS=7
NEWLINE=4
ID=5
'(' =12
')' =13
'*' =11
'=' =8
'-' =10
'+' =9
```

2.3. Prueba del Reconocedor

La ejecución de ANTLR en la gramática sólo genera el analizador léxico y el analizador sintáctico, y ExprParser ExprLexer. Para realmente tratar de la gramática en alguna entrada, tenemos un banco de pruebas con un método main () como este:

```
import org.antlr.runtime.*;
public class Test {
        public static void main(String[] args) throws Exception {
        // Create an input character stream from standard in
```

```
ANTLRInputStream input = new ANTLRInputStream(System.in);
        // Create an ExprLexer that feeds from that stream
        ExprLexer lexer = new ExprLexer(input);
        // Create a stream of tokens fed by the lexer
CommonTokenStream tokens = new CommonTokenStream(lexer);
        // Create a parser that feeds off the token stream
        ExprParser parser = new ExprParser(tokens);
        // Begin parsing at rule prog
        parser.prog();
        }
}
```

Las clases ANTLRInputStream y CommonTokenStream son clases estándar de ANTLR que se encuentran en el paquete de org.antlr.runtime. ANTLR genera todas las demás instancias de clases en el banco de pruebas. Una vez que ha compilado el código generado y el banco de pruebas, Test.java, Prueba el funcionamiento, y el tipo de una expresión simple seguida de salto de línea y el carácter de fin de archivo apropiado para su plataforma:

```
( $ javac Test.java ExprLexer.java ExprParser.java
( $ java Test
( (3+4)*5
( EOF
) $
```

ANTLR no emite salida, ya que no hay acciones en la gramática. Sin embargo, si usted escribe una expresión invalida que no está en la gramática, el analizador generado por ANTLR emitirá un error.

El analizador sintáctico también tratará de recuperarse y continuar con las expresiones correspondientes. Lo mismo puede decirse del léxico generado. Por ejemplo, al ver @ carácter no válido, el lexer informa lo siguiente:

```
( $ java Test
( 3+@
( EOF
) line 1:2 no viable alternative at character '@'
line 1:3 no viable alternative at input '\n'
$
```

El reconocedor encontró dos errores en este caso. El primer error es un error léxico: un carácter no válido, @. El segundo error es un error sintáctico: la falta de un atom (el analizador sintáctico dice 3 + \ n, un atom inválido, como un número entero).

Un analizador léxico o el analizador sintáctico emite la frase "no hay alternativa viable" cuando no se puede averiguar qué hacer cuando se enfrenta a una lista de alternativas. Esto significa que los símbolos de entrada al lado no parecen encajar en cualquiera de las alternativas.

Para los tokens que no coinciden, los reconocedores para indicar el símbolo incorrecto se encuentra en el flujo de entrada y el símbolo de espera:

```
( $ java Test
( (3
( EOF
) line 1:2 mismatched input '\n' expecting ')'
$
```

El mensaje de error dice que el token de línea nueva fue inesperada y que el analizador sintáctico espera) en su lugar. El prefijo 1:2 indica que el error ocurrió en la línea 1 y en la posición del carácter 2 dentro de esa línea. Debido a la posición del carácter comienza en 0, la posición de 2 significa que es el tercer carácter.

En este momento tenemos un programa que acepte una entrada válida o rechaza si le damos entrada inválida.

2.4. Usar la Sintaxis para la Unidad de Ejecución de la Acción

Hasta este punto se ha logrado el analizador léxico y el analizador sintáctico, todo sin necesidad de escribir código Java. Como se puede ver, escribir una gramática es mucho más fácil que escribir su propio código de análisis. Se puede pensar que la notación de ANTLR es como un lenguaje de dominio específico diseñado específicamente para que los reconocedores y traductores sean fáciles de construir.

Para pasar de un reconocedor a un traductor o intérprete, tenemos que añadir las acciones de la gramática, pero ¿qué acciones y dónde? Para nuestros propósitos, vamos a necesitar llevar a cabo las siguientes acciones:

1. Definir una tabla hash llamado de memoria para almacenar un mapa de la variable-valor.
2. Tras la expresión, imprimir el resultado de su evaluación.
3. Tras la asignación, evaluación de la expresión del lado derecho, y el mapa de la variable en el lado izquierdo con el resultado. Almacenar los resultados en la memoria.
4. En INT, devolver su valor entero como resultado.
5. En ID, devuelva el valor almacenado en la memoria para la variable. Si la variable no se ha definido, emite un mensaje de error.
6. Tras la expresión entre paréntesis, devuelva el resultado de la expresión anidada como resultado.
7. Tras la multiplicación de dos átomos, retornar la multiplicación de los resultados de los dos átomos.
8. Tras la adición de dos subexpresiones de multiplicación, retornar la adición de los resultados de las dos subexpresiones.
9. Sobre la sustracción de dos subexpresiones de multiplicación, retornar la resta de los resultados de las dos subexpresiones.

Ahora sólo hay que poner en práctica estas acciones en Java y colocarlos en la gramática de acuerdo a la ubicación que implica la vez. Comenzar por definir la memoria utilizada para asignar variables a sus valores (acción 1):

```
@header {
        import java.util.HashMap;
}
@members {
        /** Map variable name to Integer object holding value */
        HashMap memory = new HashMap();
}
```

No se necesita una acción de la regla prog, ya que sólo le dice al intérprete que busque una o más construcciones de stat. Acciones 2 y 3 de la lista detallada anterior deben ir en stat para imprimir y guardar los resultados de la expresión:

```
prog: stat+ ;
stat:  // evaluate expr and emit result
         // $expr.value is return attribute 'value' from expr call
     expr NEWLINE {System.out.println($expr.value);}
       // match assignment and stored value
// $ID.text is text property of token matched for ID reference
| ID '=' expr NEWLINE
  {memory.put($ID.text, new Integer($expr.value));}
    // do nothing: empty statement
       | NEWLINE
  ;
```

De las reglas implicadas en la evaluación de expresiones, es muy conveniente que devuelvan el valor de la subexpresión que coincida. Por lo tanto, cada regla coincide y evaluar una parte de la expresión, devolviendo el resultado como un

valor devuelto del método. Mira en el átomo, la más simple subexpresión en primer lugar:

```
atom returns [int value]
        : // value of an INT is the int computed from char sequence
        INT {$value = Integer.parseInt($INT.text);}
        |       ID // variable reference
        {
                // look up value of variable
                Integer v = (Integer)memory.get($ID.text);
                // if found, set return value else error
                if ( v!=null ) $value = v.intValue();
                else System.err.println("undefined variable "+$ID.text);
        }
                // value of parenthesized expression is just the expr value
        |       '(' expr ')' {$value = $expr.value;}
        ;
```

Por la cuarta acción de la lista detallada anteriormente, el resultado de un atom de INT es el valor entero del texto de la ficha del INT. Un token de INT con el texto 91 da como resultado el valor 91. Acción 5 nos dice que debemos buscar el texto de la ficha ID en el mapa de memoria para ver si tiene un valor.
Si es así, devolver el valor entero almacenado en el mapa, o bien imprimir un error. Rule La Tercera alternativa de la regla del átomo invoca recursivamente la regla expr. No hay nada que calcular, por lo que el resultado de esta evaluación atom es sólo el resultado de la llamada expr (); esto satisface la acción 6. Como puede ver, $ valor es la variable de resultado tal como se define en la cláusula de retorno; $ expr.value es el resultado computado por una llamada a expr. Pasando a subexpresiones de multiplicación, aquí está la regla de multExpr:

```
/** return the value of an atom or, if '*' present, return
* multiplication of results from both atom references.
```

```
* $value is the return value of this method, $e.value
* is the return value of the rule labeled with e.
*/
multExpr returns [int value]
   : e=atom {$value = $e.value;} ('*' e=atom {$value *= $e.value;})*
   ;
```

Regla multExpr coincide con un atom puede ir seguido por una secuencia de operadores y operandos * atom. Si no hay ningún operador * tras el primer átomo, entonces el resultado de multExpr es sólo resultado de atom. Para cualquier multiplicación que siguen del primer atom, todo lo que tenemos que hacer es mantener la actualización de los resultados multExpr, $ valor, por la acción 7. Cada vez que se observa un * y un atom, se multiplica el resultado por el resultado multExpr. Las acciones de la regla expr, la regla de la expresión exterior, refleja las acciones en multExpr excepto que estamos sumando y restando en lugar de multiplicar:

```
/** return value of multExpr or, if '+'|'-' present, return
* multiplication of results from both multExpr references.
*/
expr  returns [int value]
      : e=multExpr {$value = $e.value;}
      ( '+' e=multExpr {$value += $e.value;}
      | '-' e=multExpr {$value -= $e.value;}
      )*
      ;
```

Estas acciones cumplen las últimas acciones, 8 y 9, de la lista detallada anteriormente en esta sección. Una de las grandes lecciones que aprender aquí es que la sintaxis conduce a la evaluación de las acciones en el analizador sintáctico. La estructura de una secuencia de entrada indica la clase de cosa que es. Por lo tanto, para ejecutar acciones sólo para una construcción particular,

todo lo que tenemos que hacer es colocar las acciones en la alternativa de la gramática que coincide con la construcción.

2.5. ¿Qué hace ANTLR con las acciones de la gramática?

ANTLR simplemente inserta acciones justo después de que genera código para el elemento anterior, los analizadores sintácticos deben ejecutar acciones incrustadas tras igualar el elemento de la gramática anterior ANTLR les arroja al pie de la letra a excepción de las traducciones de referencia de atributo especiales y la plantilla. El manejo de las acciones de ANTLR es muy sencillo. Por ejemplo, ANTLR traduce las reglas retornando especificaciones como el siguiente:

```
multExpr returns [int value]
: ...
;
```

Para el siguiente código Java:

```
public int multExpr() throws RecognitionException {
        int value = 0; // rule return value, $value
        ...
        return value;
}
```

ANTLR traduce las etiquetas de referencias regla, tales como e = multExpr, a las asignaciones de llamada al método, tales como e = multExpr Las referencias a la regla de los valores de retorno, tal como $ e.value, se convierten en e cuando sólo hay un valor de retorno y e.value cuando hay varios valores de retorno.
Podemos observar el pseudocódigo para expr regla. Las líneas de relieve en la producción se derivan de acciones incrustadas:

```
Public int expr() {
        int value = 0; // our return value, automatically initialized
        int e = 0;
        try {
                e=multExpr();
                value = e; // if no + or -, set value to result of multExpr
                // Expr.g:27:9: ( '+' e= multExpr | '-' e= multExpr )*
                loop3:
                while ( true ) {
                int alt=3;
                if ( «next input symbol is +» ) { alt=1; }
                else if ( «next input symbol is -» ) { alt=2; }
                switch ( alt ) {
                case 1 :
                        match('+' );
                        e=multExpr();
                        value += e; // add in result of multExpr
                        break;

                case 2 :
                        match('-' );
                        e=multExpr();
                        value -= e; // subtract out result of multExpr
                        break;
                default :
                        break loop3;
                }
        }
}
catch (RecognitionException re) {
        reportError(re);
        recover(input,re);
```

```
}
return value;
}
```

Ahora se puede poner a prueba la gramática. Hemos añadido las acciones sólo a la gramática, por lo que el programa main () en la prueba puede permanecer igual. Todavía sólo invoca la regla de empezar prog.Debemos tener en cuenta que si cambiamos la gramática, tenemos que volver a compilar los archivos generados, tales como ExprParser.java y ExprLexer.java. Ejecución de las expresiones en el programa devuelve ahora los cálculos esperados:

```
( $ java Test
( 3+4*5
( EOF
) 23
( $ java Test
( (3+4)*5
( EOF
) 35
$
```

Las asignaciones de variables almacenan resultados de la expresión, y luego se puede sacar los resultados más tarde haciendo referencia a la variable, como se muestra en el archivo de entrada:

```
a=3
b=4
2+a*b
```

Ejecución de las expresiones en el banco de pruebas con la redirección de IO da el resultado correcto:

```
$ java Test < input
14
$
```

Tras la entrada no válida, ANTLR informa de un error e intenta recuperar. Recuperarse de un error significa volver a sincronizar el analizador sintáctico y pretender que no pasó nada. Esto significa que el intérprete todavía debe ejecutar acciones después de recuperarse de un error. Por ejemplo, si escribe 3++4, el analizador de falla en el + segundo porque no puede hacer coincidir con un atom:

```
( $ java Test
( 3++4
( EOF
) line 1:2 no viable alternative at input '+'
7
$
```

El analizador sintáctico se recupera echando tokens hasta que ve a un atom de validez, que es de 4 en este caso. También puede recuperar de los símbolos que faltan por pretender insertarlos. Por ejemplo, si se deja fuera de un paréntesis, el analizador sintáctico informa de un error, pero luego continúa como si hubiera escrito el):

```
( $ java Test
( (3
( EOF
) line 1:2 mismatched input '\n' expecting ')'
3
$
```

Con esto concluye el primer ejemplo ANTLR completo. El programa utiliza una gramática para que coincida con las expresiones y acciones incrustadas que utiliza para evaluar expresiones. Ahora podemos analizar una solución más sofisticada para el mismo problema. La solución es más complicada, pero vale la pena, ya que muestra cómo crear una estructura de datos del árbol y caminar con otra gramática. Esta estrategia multipaso es útil para traducciones complicadas porque son mucho más fáciles de manejar si se rompen en varios fragmentos simples.

2.6. Evaluación de Expresiones Mediante la forma de AST Intermedio

Ahora se ha visto cómo construir una gramática y agregar acciones para implementar una traducción, esta sección le guiará a través de la construcción de la misma funcionalidad pero con un paso adicional, con árboles. Se va a usar la misma gramática del analizador sintáctico para construir una estructura de datos intermedia, en sustitución de las acciones integradas con las normas de construcción del árbol. Una vez que tengamos ese árbol, se va a utilizar un analizador sintáctico de árbol para recorrer el árbol y ejecutar acciones incrustadas.

ANTLR va a generar un árbol sintáctico a partir de un árbol gramatical automáticamente para nosotros. La gramática analizador convierte una cadena de componentes léxicos en un árbol que el árbol gramatical analiza y evalúa. Aunque el enfoque de la sección anterior, era más sencillo, no se adapta bien a un lenguaje de programación. Añadiendo construcciones tales como las llamadas a funciones o bucles while con el lenguaje significa que el intérprete debe ejecutar los mismos bits de código de tiempos múltiples. Este enfoque funciona, pero no es tan flexible como la construcción de una representación intermedia como un árbol de sintaxis abstracta (AST) y luego recorrer esa estructura de datos para interpretar las expresiones y asignaciones. Varias veces a pie de un árbol de forma intermedia es mucho más rápido que reanalizar un programa de entrada. Una representación intermedia suele ser un árbol de un poco de gusto y

los registros no sólo los símbolos de entrada, sino también la relación entre los símbolos según lo dictado por la estructura gramatical. 3+4: Por ejemplo, el AST siguiente representa la expresión 3 +4:

Figura 4. AST para expresión 3+4

En muchos casos, se verá los árboles representados en forma de texto. Por ejemplo, la representación de texto 3 +4 es (+ 3 4). El primer símbolo después del (es la raíz y los símbolos subsiguientes son sus hijos el AST para la expresión AST 3+4 * 5 tiene la forma de texto (+ 3 (* 4 5)) y se ve así:

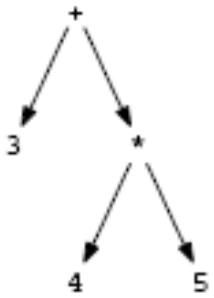


Figura 5. AST para expresión 3+4*5

Como se puede ver, la estructura del árbol implícitamente codifica la precedencia de los operadores. En este caso, la multiplicación se debe hacer en primer lugar porque la operación de suma necesita el resultado de la multiplicación como operador de la derecha.

Un AST se diferencia de un árbol de análisis, el que representa la secuencia de invocaciones a la regla utilizada para que coincida con un flujo de entrada. La Figura a continuación muestra el árbol de análisis para 3 +4 (creado por ANTLRWorks).

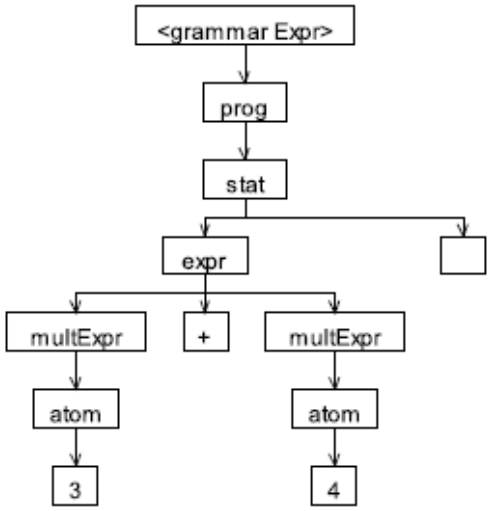


Figura 6. Árbol de Análisis para 3+4 creado por ANTLRWorks

Las hojas de un árbol de análisis son los símbolos de entrada, y las no-hojas son los nombres de las reglas (el nodo más alto, <grammarExpr>, es algo que ANTLRWorks agrega para mostrar lo que el árbol de análisis gramatical trae). El nodo de la regla superior, prog, indica que 3 +4 es un prog en general. Más específicamente, es una stat, que a su vez es una expr seguido por un salto de línea, y así sucesivamente. Por lo tanto los registros de árbol de análisis sintáctico como el reconocedor, navegan por las reglas de la gramática para que coincida con la entrada. Podemos comparar esto con el AST mucho más pequeño y más simple para 3 +4, donde todos los nodos del AST son nodos de entrada de símbolos. La estructura del árbol codifica el significado que '+' es un operador con dos hijos. Esto es mucho más fácil de ver sin todo el "ruido" introducido por los nodos de regla gramatical.

En la práctica, también es útil para separar la gramática de los árboles que se produce, por lo que AST son superiores a analizar los árboles. A modificar en una gramática general, altera la estructura de un árbol de análisis, dejando afectado un AST, que puede hacer una gran diferencia en el código que recorre sus árboles.

Una vez que tenga el árbol, se puede recorrer en múltiples formas con el fin de evaluar el árbol que representa la expresión. En general, se recomienda el uso

de una gramática para describir la estructura del árbol del mismo modo que el uso de una gramática sintáctica para describir un idioma de introducción de una dimensión. Del árbol de la gramática, ANTLR puede generar un método para recorrer el árbol usando la misma forma de arriba a abajo por descenso recursivo estrategia de análisis utilizados para analizadores léxicos y sintácticos. Más adelante aprenderemos cómo construir AST, la forma de caminar con una gramática de árbol, y la manera de integrar las acciones dentro de una gramática de árbol para emitir una traducción. Al final, se obtendrá un traductor que es funcionalmente equivalente a la anterior.

2.7. Construir un AST con una gramática

La construcción de AST con ANTLR es sencilla. Sólo se tiene que añadir reglas de construcción AST a la gramática del analizador sintáctico que indican la forma del árbol que queremos construir. Este enfoque declarativo es mucho más pequeño y más rápido a leer y escribir que la alternativa informal de la utilización de acciones arbitrarias integrados. Cuando se utiliza la opción de output = AST, cada una de las reglas de la gramática implícita devolverá un nodo o subárbol. El árbol que se obtiene de la invocación de la regla de partida es el AST completo. Tomemos la gramática del analizador sintáctico sin acciones, el reconocimiento de la sintaxis del lenguaje anterior y lo aumentamos para construir un AST adecuado. A medida que avanzamos, vamos a discutir la estructura AST apropiado. Empieza por contarnos ANTLR para construir un nodo de árbol para cada token emparejado en el flujo de entrada.

```
grammar Expr;
options {
        output=AST;
        // ANTLR can handle literally any tree node type.
        // For convenience, specify the Java type
        ASTLabelType=CommonTree; // type of $stat.tree ref etc...
}
```

Para cada token que coincide con el reconocedor, se creará un único nodo de AST. Dado instrucciones en contrario, el reconocedor construye un árbol de plano (una lista enlazada) de los nodos. Para especificar una estructura de árbol, se limitan a indicar que los tokens deben ser considerados operadores (raíces subárbol) y que los tokens deben ser excluidos de los árboles. Utilice ^ y ! token de referencia y sufijo, respectivamente.
A partir de la gramática del analizador sintáctico sin acciones, modificar las reglas de expresión de la siguiente manera:

```
expr: multExpr (('+' ^|'-' ^) multExpr)*
;
multExpr
: atom ('*' ^ atom)*
;
atom:          INT
            | ID
            | '(' ! expr ')' !
;
```

Tan sólo hay que añadir los operadores de AST a los tokens +, -, *, (and). El sufijo ! operador en el paréntesis indica a ANTLR que debe evitar la construcción de los nodos de los tokens. Los paréntesis alteraran la normal prioridad de los operadores, cambiando el orden de las llamadas a reglas de métodos. La estructura del árbol generado, por lo tanto, codifica el operador aritmético de prioridad, por lo que no necesita paréntesis en el árbol.

Para la regla de prog y stat, vamos a utilizar la sintaxis de árboles reescribir ya que es más clara. Para cada alternativa, añada a -> regla de construcción AST de la siguiente manera:

```
/** Match a series of stat rules and, for each one, print out the
```

```
* tree stat returns, $stat.tree. toStringTree() prints the tree
* out in form: (root child1 ... childN). ANTLR's default tree
* construction mechanism will build a list (flat tree) of the stat
* result trees. This tree will be the input to the tree parser.*/
prog: ( stat {System.out.println(
$stat.tree==null?"null":$stat.tree.toStringTree());} )+ ;
stat: expr NEWLINE -> expr
| ID '=' expr NEWLINE -> ^('=' ID expr)
| NEWLINE ->
;
```

Los elementos de la gramática a la derecha del operador -> son fragmentos de árboles gramaticales que indican la estructura del árbol que queremos construir. El primer elemento dentro de una especificación de árboles ^(...) es la raíz del árbol. Los elementos restantes son hijos de esa raíz. Usted puede pensar en las reglas de reescritura como las transformaciones de la gramática a la gramática. Ya veremos en un momento en que esas reglas exactas de la construcción del árbol convertido en alternativas de la gramática de los árboles. La regla prog sólo imprime los árboles y, además, no necesita la construcción del árbol explícito. El árbol de comportamiento por defecto de construcción para prog construye lo que quiere: una lista de árboles declaración a analizar con la gramática de los árboles.

La regla de reescritura de la primera alternativa, dice que el valor estadístico de retorno es el árbol devuelto desde la llamada expr. La segunda alternativa de reescritura dice que para construir un árbol con '=' en la raíz y la identificación como el primer hijo. El árbol devuelto desde la llamada expr es el segundo hijo. La reescritura de vacío para la tercera alternativa, simplemente significa no crear un árbol. Las reglas léxicas y el programa principal en la prueba no es necesario ningún cambio.

Vamos a ver lo que el traductor hace con el archivo anterior de entrada:

```
a=3
```

```
b=4
2+a*b
```

En primer lugar decir a ANTLR que traduzca Expr.g a código Java y se compile como lo hizo con la solución anterior:

```
$ java org.antlr.Tool Expr.g
ANTLR Parser Generator Version 3.0 1989-2007
$ javac Test.java ExprParser.java ExprLexer.java
$
```

Ahora, redirigir la entrada del archivo en el banco de pruebas, para poder ver tres árboles impresos, una para cada asignación de entrada o de expresión. El banco de pruebas imprime el árbol que devuelve de la regla del comienzo prog en forma de texto:

```
$ java Test < input
(= a 3)
(= b 4)
(+ 2 (* a b))
$
```

El AST completo construido por el analizador sintáctico (y devuelto de prog) tiene el siguiente aspecto en la memoria:

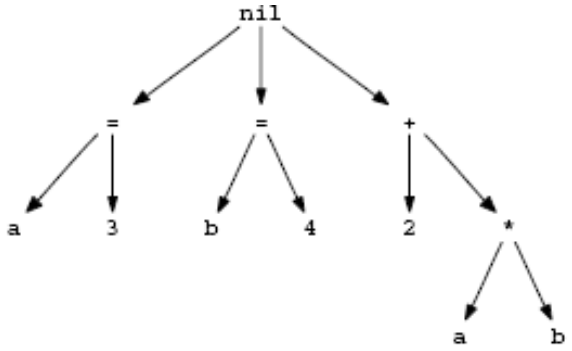


Figura 7. AST construído por el Analizador Sintáctico

El nodo nulo representa una lista de subárboles. Los hijos del nodo nulo siguen de los elementos de una lista.

Ahora tenemos un programa de análisis que se basa AST apropiados, y necesitamos una manera de recorrer los árboles para evaluar las expresiones que representan. La sección siguiente muestra cómo construir una gramática de árbol que describe la estructura de AST y la manera de integrar las acciones, mediante la sintaxis para impulsar la ejecución de la acción.

2.8. Evaluación de expresiones codificadas en AST

En esta sección, se escribe un árbol gramatical para describir la estructura del AST que se construyó utilizando una gramática del analizador en la sección anterior. Entonces, debemos agregar acciones para calcular los resultados subexpresión. Al igual que la solución anterior, cada regla de expresión devolverá estos resultados parciales. A partir de esta gramática, ANTLR construirá un analizador de árboles que ejecuta sus acciones incrustadas. Analizar un árbol es una cuestión de recorrerlo, y verificar no sólo los nodos adecuados, sino también la propia estructura de dos dimensiones. Ya que es más difícil construir analizadores que reconocer directamente las estructuras de árbol, ANTLR utiliza una corriente unidimensional de los nodos del árbol, calcula iterando sobre los nodos de un árbol a través de un recorrido en profundidad. ANTLR inserta nodos especiales imaginarios arriba y abajo para indicar que el árbol original se dejó caer a una lista de elementos secundarios o terminados para recorrer una lista de elementos secundarios. De esta manera, reduce el análisis de ANTLR convencional de una dimensión de análisis cadena de componentes léxicos. Por ejemplo, la tabla siguiente resume cómo ANTLR serializa dos árboles de entrada.

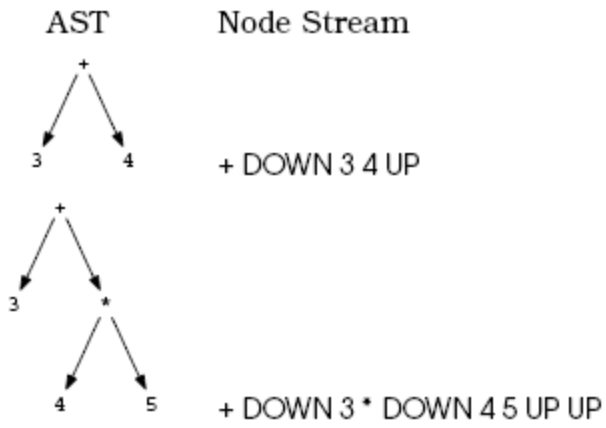


Figura 8. Evaluación de expresiones en AST

La notación ANTLR para una gramática de árbol es idéntico a la notación de una gramática regular a excepción de la introducción de una construcción del árbol de dos dimensiones. Lo bueno de esto es que podemos hacer una gramática de árboles por cortar y pegar de la gramática del analizador sintáctico. Sólo tenemos que eliminar los elementos de gramática de reconocimiento a la izquierda del operador ->, dejando a los fragmentos de reescribir AST. Estos fragmentos de construir AST en la gramática del analizador y reconocer la estructura de la gramática de los árboles.

Vamos a construir un árbol gramatical en un archivo separado, al que llamaremos Eval.g. Los árboles gramaticales comienzan muy parecido a las gramáticas sintácticas con una cabecera de gramática y algunas opciones:

```
tree grammar Eval; // yields Eval.java
options {
        tokenVocab=Expr; // read token types from Expr.tokens file
        ASTLabelType=CommonTree; // what is Java type of nodes?
        }
```

La opción tokenVocab indica que la gramática del árbol debe precargar los nombres de token y se asocia tipos de tokens definidos en Expr.tokens. (ANTLR genera este archivo después de la elaboración Expr.g.) Cuando hablamos de ID

en la gramática de árboles, queremos que el reconocedor como resultado de usar el mismo tipo señal que el analizador utiliza. ID en el árbol de análisis debe coincidir con el mismo tipo de símbolo que lo hizo en el analizador sintáctico.
Antes de escribir las reglas, definir una tabla hash de memoria para almacenar los valores de variables, como lo hicimos en la solución de la gramática del analizador sintáctico:

```
@header {
import java.util.HashMap;
}
@members {
/** Map variable name to Integer object holding value */
HashMap memory = new HashMap();
}
```

La estructura adecuada AST, debe simplificar y normalizar las versiones de la cadena de componentes léxicos que implícitamente codifica la estructura gramatical. En consecuencia, los árboles gramaticales suelen ser mucho más simples que el analizador de gramáticas asociadas que construyen sus árboles. De hecho, en este caso, todas las reglas de expresión a partir del colapso de gramática del analizador sintáctico a una simple regla expr en la gramática de árboles. Esta gramática del analizador sintáctico normaliza los árboles de expresión que tiene un operador en la raíz y los dos operandos como sus hijos. Necesitamos una regla expr que refleja esta estructura:

```
expr returns [int value]
        : ^('+' a=expr b=expr) {$value = a+b;}
        | ^('-' a=expr b=expr) {$value = a-b;}
        | ^('*' a=expr b=expr) {$value = a*b;}
        | ID
        {
        Integer v = (Integer)memory.get($ID.text);
```

```
if ( v!=null ) $value = v.intValue();
else System.err.println("undefined variable "+$ID.text);
}
| INT {$value = Integer.parseInt($INT.text);}
;
```

La regla expr indica que un árbol de expresión es un nodo simple creado a partir de una identificación o un token de INT, o que un árbol de expresión es un operador subárbol. Con la simplificación de la gramática viene una simplificación de las acciones asociadas. La regla expr normaliza todos los cálculos a ser de la forma "número uno = b. <operador>" Las acciones de ID y los nodos de INT son idénticas a las acciones que se utilizó en regla atom de la gramática del analizador sintáctico. Las acciones de las reglas prog y stat son idénticas a la solución anterior. Regla prog no tiene una acción que sólo coincide con una secuencia de expresión o de asignación de los árboles. Regla stat hace una de dos cosas:

1. Que coincida con una expresión e imprime el resultado.
2. Que coincida con una asignación de mapas y el resultado a la indicada variable.

Así es como debe decir en la notación de ANTLR:

```
prog: stat+ ;
stat: expr
{System.out.println($expr.value);}
| ^('=' ID expr)
{memory.put($ID.text, new Integer($expr.value));}
;
```

Regla STAT no tiene una tercera alternativa para que coincida con la nueva línea (vacío) como expresión de la solución anterior. El analizador sintáctico excluye las expresiones vacías para no construir los árboles de ellos.

¿Qué pasa con las reglas léxicas? Resulta que no es necesario porque cualquier árbol gramatical se alimenta de una corriente de los nodos del árbol, no tokens.
En este punto, tenemos una gramática que construye un analizador de AST y un árbol gramatical que reconoce la estructura de árbol, la ejecución de acciones para evaluar expresiones. Antes de que podamos probar las gramáticas, tenemos que preguntarnos si ANTLR traduce la gramática Eval.g a código Java. Ejecutar el siguiente comando:

```
$ java org.antlr.Tool Eval.g
ANTLR Parser Generator Version 3.0 1989-2007
$
```

Esto se traduce en los siguientes dos archivos:

Tabla 5. Archivos que genera ANTLR (Código Java)

Archivo Generado	Descripción
Eval.java	El descenso recursivo del árbol sintáctico generado por la gramática.
Eval.tokens	La lista de, nombre del token y el tipo de token, asignaciones como INT = 6. Nadie va a utilizar este archivo, en este caso, pero siempre ANTLR genera un archivo de vocabulario simbólico.

Ahora se tiene que modificar el banco de pruebas para que recorrer el árbol construido por el intérprete. En este punto, lo único que hace es lanzar el analizador sintáctico, por lo que debe agregar código para extraer el árbol de resultados del analizador, creación de un método que recorra los árboles de tipo Eval, y empezar a recorrer el árbol con la regla prog. Aquí está el banco de pruebas completo que hace todo lo que necesitamos:

```
import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;
public class Test {
        public static void main(String[] args) throws Exception {
        // Create an input character stream from standard in
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        // Create an ExprLexer that feeds from that stream
        ExprLexer lexer = new ExprLexer(input);
        // Create a stream of tokens fed by the lexer
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        // Create a parser that feeds off the token stream
        ExprParser parser = new ExprParser(tokens);
        // Begi          n parsing at rule prog, get return value structure
        ExprParser.prog_return r = parser.prog();
        // WALK RESULTING TREE
        CommonTree t = (CommonTree)r.getTree(); // get tree from parser
        // Create a tree node stream from resulting tree
CommonTreeNodeStream nodes = new CommonTreeNodeStream(t);
        Eval walker = new Eval(nodes); // create a tree parser
        walker.prog(); // launch at start rule prog
        }
}
```

El banco de pruebas extrae del AST en el analizador sintáctico para conseguir los beneficios de valor de retorno prog de objetos. Este objeto es del tipo prog_return, que ANTLR genera en ExprParser:

```
// from the parser that builds AST for the tree grammar
public static class prog_return extends ParserRuleReturnScope {
        CommonTree tree;
public Object getTree() { return tree; }
        };
```

En este caso, en la regla prog no se ha definido por el usuario un valor de retorno, por lo que el árbol construido es el único valor de retorno. Se tiene un completo evaluador de expresiones ahora. Por ejemplo se ingresa algunas expresiones a través de la entrada estándar:

```
( $ java Test
( 3+4
( EOF
) (+ 3 4)
7
( $ java Test
( 3*(4+5)*10
( EOF
) (* (* 3 (+ 4 5)) 10)
270
$
```

También se puede redirigir el archivo de entrada en el banco de pruebas:

```
$ java Test < input
(= a 3)
(= b 4)
(+ 2 (* a b))
14
$
```

La salida de la instalación de la primera prueba muestra la estructura de árbol (en forma de serie) por cada instrucción de entrada. Forma de serie (= a 3) representa el árbol construido para a = 3. El árbol tiene = en la raíz y sus dos hijos: uno y tres. El equipo emite entonces el valor de expresión calculada por el analizador de los árboles. En este capítulo, se ha construido dos evaluadores de expresiones equivalentes. La primera implementación evaluó expresiones

directamente en una gramática del analizador sintáctico, que funciona muy bien para las más simples traducciones y es la forma más rápida de construir un traductor. La segunda implementación separó el análisis y la evaluación en dos fases. La primera fase se analiza como antes, pero construye AST en lugar de evaluar las expresiones de inmediato. La segunda fase recorre los árboles resultantes para hacer la evaluación. Se necesitará este segundo enfoque en la construcción de traductores complicados, que son más fáciles de entender cuando se descomponen en subproblemas.

De hecho, algunas tareas de lenguaje, tales como intérpretes de lenguaje de programación, tienen que recorrer varias veces la entrada por su propia naturaleza. La construcción de una forma intermedia simple y concisa es mucho más rápido que en repetidas ocasiones el análisis de la cadena de componentes léxicos. En este punto, se tiene una idea general de cómo trabajar con ANTLR y cómo escribir gramáticas simples para reconocer y traducir frases de entrada. [8]

[8] PARR, Terence. 2010, A quick tour for the impatient, EN: The Definitive ANTLR References, Building Domain-Specifiv Language, Dallas, Texas, pp. 59-84.

E. METODOLOGÍA

E.1. Métodos

E.1.1. Método Inductivo

Este método se aplica para determinar cuáles son los problemas que tienen los alumnos de Cuarto y Quinto Módulo de la carrera de Ingeniería en Sistemas en la Unidad de Metodología de la Programación, a través de la recolección de información con la aplicación de la técnica de entrevista estructurada dirigida a los estudiantes, quienes serán los usuarios de la aplicación.

E.1.2. Método Deductivo

Una vez que conocemos los problemas que existen en la Unidad de Metodología de la Programación, se aplica el método deductivo para determinar causa el origen de estos problemas a través de la tabulación de las entrevistas realizadas a los estudiantes, con estas estadísticas podemos concluir las causas que originan el problema y su incidencia, para luego con el análisis, plantear alternativas de solución.

E.2. Técnicas

E.2.1. Entrevista estructurada

Se utiliza la entrevista para conocer cuáles son los problemas que tienen los alumnos de cuarto y quinto módulo de la carrera de Ingeniería en Sistemas, a quienes se les realizó la entrevista.

E.3. Metodología Para Desarrollo Del Software

Para obtener un software de calidad se ha creído conveniente utilizar una combinación de SCRUM y XP, ya que por sus características se adaptan a las necesidades.

En primera instancia se utiliza la metodología SCRUM es un método ágil para el desarrollo de software que deja mucho a la opinión del equipo de desarrollo de software, en la metodología Scrum los roles se definen de la siguiente manera:

Tabla 6. Roles en SCRUM

ScrumMaster	Director de proyecto (Ing. Edison Coronel), el cual se encarga de la supervisión del cumplimiento de los objetivos o metas.
ProductOwner	Usuario (Docente y Alumnos del quinto Módulo de la carrera de Ingeniería en Sistemas), es quien determina la funcionalidad que debe tener la aplicación, para ello se le realiza una entrevista en la que expone todos sus requerimientos.
Team	Desarrolladores (Alex Román y Alondra Ordóñez), son los encargados de la implementación de la aplicación tal y como el ProductOwner disponga en sus requerimientos, y bajo la supervisión del ScrumMaster.

Ademas la metodología SCRUM ha permitido llevar el desarrollo del proyecto a través de fases llamadas Sprints, las cuales se detallan a continuación:

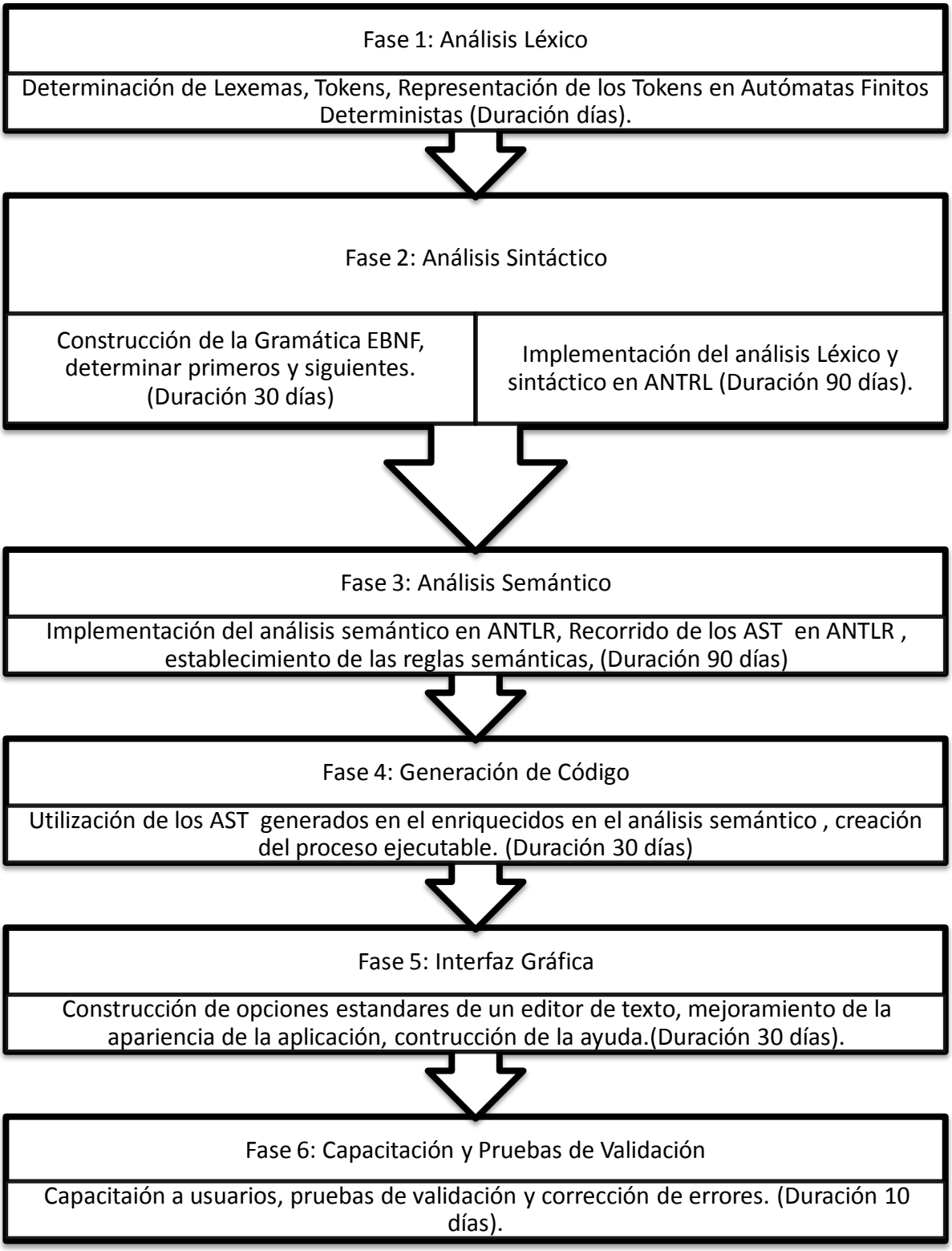


Figura 9. Sprints de SCRUM

Se acompaña de la metodología XP que es una disciplina de desarrollo de software que se lleva a cabo con prácticas simples y con suficiente información para que el equipo pueda ver cómo avanza y pueda ajustar las prácticas a su situación particular, dentro de la información necesaria para XP se ha determinado una historia de usuario sin perder el tiempo en cosas irrelevantes como demasiada documentación por ejemplo.

Scrum y XP (Programación eXtrema) se combinan de forma fructífera, por un lado Scrum ayuda a enfocarse en las prácticas de organización y gestión, mientras que XP se centra más en las prácticas de programación. Por ésta razón han funcionado tan bien juntas durante el desarrollo de la aplicación ya que tratan áreas diferentes pero se complementan entre ellas.

F. RESULTADOS

F.1. Desarrollo de la Propuesta Alternativa

F.1.1. Análisis Léxico

F.1.1.1. Lexemas y Tokens

Lexemas son el conjunto de palabras reservadas, signos, operadores y separadores del Lenguaje ARK, las palabras reservadas son palabras clave que no se pueden utilizar como identificadores, los signos y operadores sirven para realizar las operaciones y los separadores nos permiten indicar dónde comienza y termina cada uno de nuestros lexemas y la gramática en sí.

Tabla 7. Lexemas y Tokens

DESCRIPCION TOKEN	N° TOKEN	LEXEMA	EXPRESION REGULAR	SEPARA-DORES
TIPOS BASICOS				
Tipo de dato entero	4	entero	entero	\n,\t,\r
Tipo de dato real	5	real	real	\n,\t,\r
Tipo de dato booleano	6	bool	bool	\n,\t,\r
Tipo de dato cadena	7	cadena	cadena	\n,\t,\r
Tipo de dato carácter	8	caracter	carácter	\n,\t,\r
PALABRAS RESERVADAS				
Constante	9	const	const	\n,\t,\r
Imprimir en	10	imprimir	imprimir	\n,\t,\r

consola				
Inicio de algoritmo	11	INICIO	INICIO	\n,\t,\r
Fin del algoritmo	12	FIN	FIN	\n,\t,\r
Condición Si	13	si	si	\n,\t,\r, (
instrucción entonces	14	entonces	entonces	\n,\t,\r, (
fin condición si	15	finsi	finsi	\n,\t,\r
Instrucción sino	16	sino	sino	\n,\t,\r
Fin de instrucción sino	17	finsino	finsino	\n,\t,\r, (
Bucle mientras	20	mientras	mientras	\n,\t,\r, (
Instrucción hacer	19	hacer	hacer	\n,\t,\r, (
Fin bucle mientras	18	finmientras	finmientras	\n,\t,\r
Instrucción hasta	21	hasta	hasta	\n,\t,\r, (
Bucle para	22	para	para	\n,\t,\r, (
Fin bucle para	23	finpara	finpara	\n,\t,\r
Instrucción seleccionar	24	seleccionar	seleccionar	\n,\t,\r, (
Caso	25	caso	caso	\n,\t,\r
Instrucción defecto	26	defecto	defecto	\n,\t,\r, {
Fin instrucción seleccionar	27	finseleccionar	finselecciona r	\n,\t,\r
Instrucción parar		parar	parar	\n,\t,\r
Inicio Función	28	funcion	funcion	\n,\t,\r, (
Fin de Función	29	finfuncion	finfuncion	\n,\t,\r
Inicio de Procedimiento	30	procedimiento	procedimient o	\n,\t,\r, (
Fin de	31	finprocedimiento	finprocedimie	\n,\t,\r

procedimiento			nto	
Instrucción retornar	32	retornar	retornar	\n,\t,\r, (
Instrucción repetir	87	repetir	repetir	\n,\t,\r, (
Instrucción hasta que		hastaque	hastaque	\n,\t,\r, (
Función Trigonométrica Seno	34	seno	sen	\n,\t,\r, (
Función Trigonométrica Coseno	33	coseno	cos	\n,\t,\r, (
Función Trigonométrica Tangente	35	tangente	tan	\n,\t,\r, (
SIGNOS				
Punto	52	.	.	IDENT
Punto y coma	53	;	;	\n,\t,\r
Coma		,	,	\n,\t,\r, IDENT
Abrir paréntesis	55	(	(	\n,\t,\r, (0-9),(0-9).(0-9), IDENT
Cerrar Paréntesis	56	)	)	\n,\t,\r, (0-9),(0-9).(0-9)
Dos puntos	57	:	:	\n,\t,\r,(
Abrir corchete	58	[	[	\n,\t,\r,(
Cerrar corchete	59	]	]	\n,\t,\r,(
OPERADORES				
Operador Igual	60	==	[=U=]	\n,\t,\r, (0-9),(0-9).(0-9), IDENT
Operador diferente	61	!=	[!U=]	\n,\t,\r, (0-9),(0-9).(0-9), IDENT

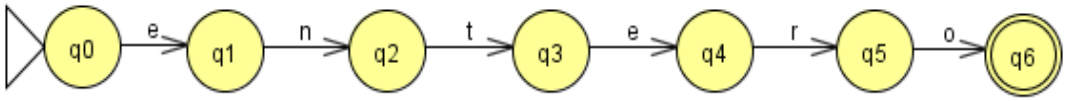
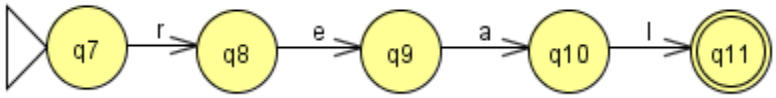
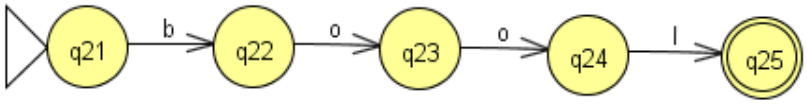
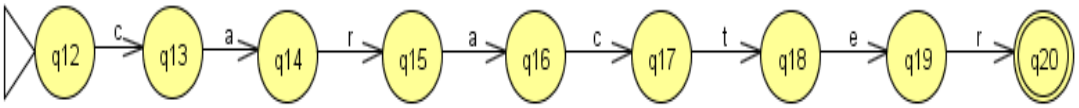
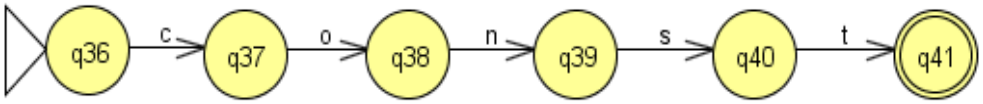
Operador asignación	62	=	=	\n,\t,\r, (0-9),(0-9).(0-9), IDENT
Operador menor que	65	<	<	\n,\t,\r, (0-9),(0-9).(0-9), IDENT
Operador mayor que	64	>	>	\n,\t,\r, (0-9),(0-9).(0-9), IDENT
Operador menor o igual	63	<=	[<U=]	\n,\t,\r, (0-9),(0-9).(0-9), IDENT
Operador mayor o igual	66	>=	[>U=]	\n,\t,\r, (0-9),(0-9).(0-9), IDENT
Operador suma	67	+	+	\n,\t,\r, (0-9),(0-9).(0-9), IDENT
Operador resta	68	-	-	\n,\t,\r, (0-9),(0-9).(0-9), IDENT
Operador producto	69	*	*	\n,\t,\r, (0-9),(0-9).(0-9), IDENT
Operador división	70	/	/	\n,\t,\r, (0-9),(0-9).(0-9), IDENT
Operador Potencia	36	pot	pot	\n,\t,\r, (0-9),(0-9).(0-9), IDENT
Operador residuo	71	%	%	\n,\t,\r, (0-9),(0-9).(0-9), IDENT
Operador raíz cuadrada	37	raíz	raiz	\n,\t,\r,(
Operador ó lógico	72	||	[|U|]	\n,\t,\r, (0-9),(0-9).(0-9), IDENT
Operador y lógico	73	&&	[&U&]	\n,\t,\r, (0-9),(0-9).(0-9), IDENT
Operador negación	74	~	~	\n,\t,\r, (0-9),(0-9).(0-9), IDENT
Operador coma	54	,	,	\n,\t,\r, IDENT

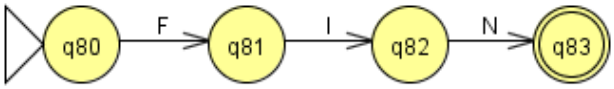
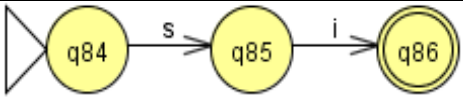
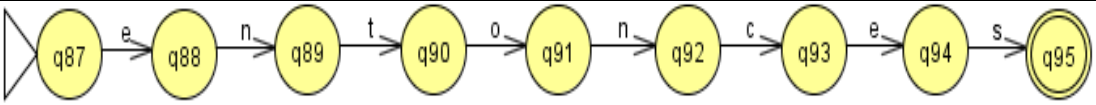
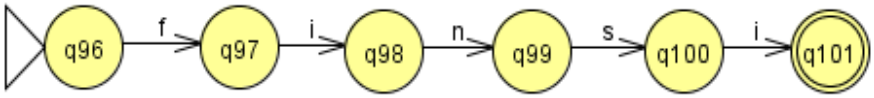
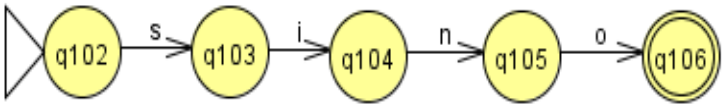
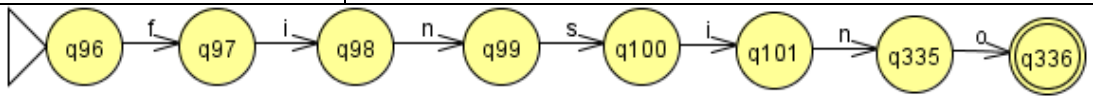
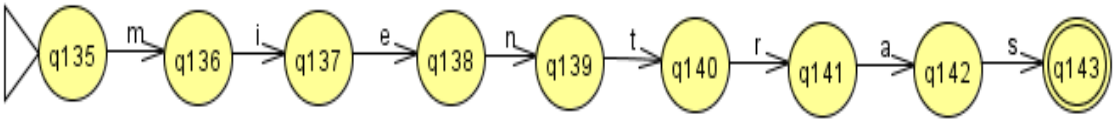
Operador más más	75	++	[+U+]	\n,\t,\r, IDENT
Operador menos menos	76	--	--	\n,\t,\r, IDENT
Operación menos Unario	51	-	-	\n,\t,\r, , (0-9),(0-9).(0-9)
Identificadores	77	IDENT	[A-Z-|a-z](A-Z-|a-z)|[0-9]|[_])*	\n,\t,\r,(
Números enteros	84	0-9	[0-9]+	\n,\t,\r,), IDENT, opeAritm, opeRelac
Números reales	79	(0-9).(0-9)	[[0-9]+(.) [0-9]+]	\n,\t,\r,), IDENT, opeAritm, opeRelac

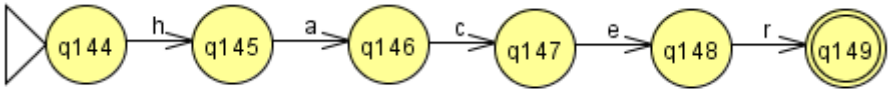
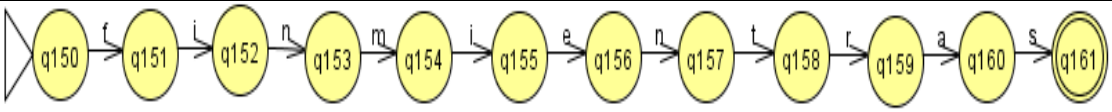
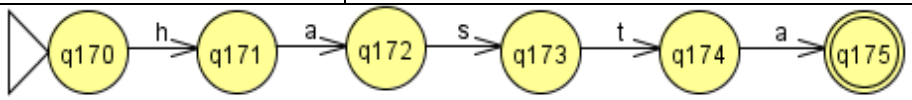
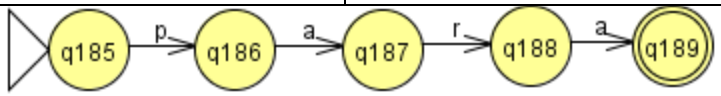
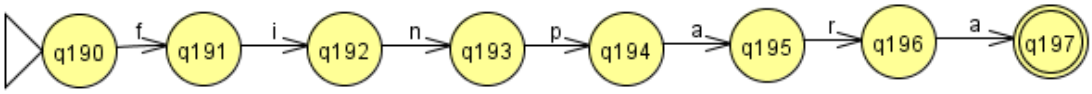
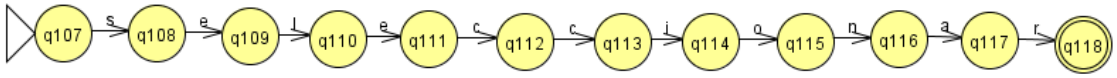
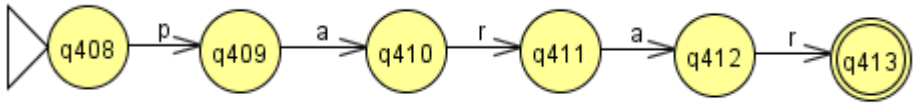
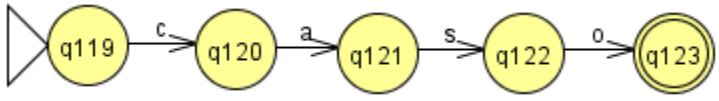
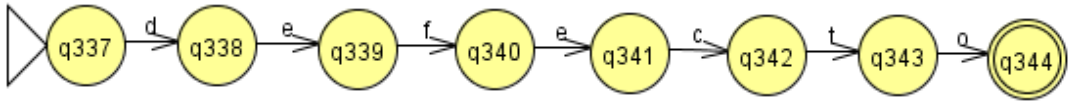
F.1.1.2. Autómatas Individuales

Los autómatas se utilizan para representar gráficamente la secuencia de caracteres de las palabras reservadas, los signos y los operadores de nuestra gramática.

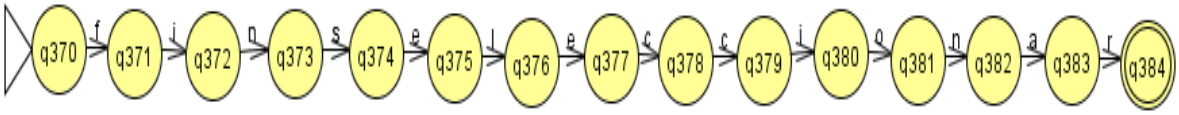
Tabla 8. Autómatas Individuales

DESCRIPCION TOKEN	N° TOKEN
Tipo de dato entero	4
	
Tipo de dato real	5
	
Tipo de dato booleano	6
	
Tipo de dato cadena	7
	
Tipo de dato carácter	8
	
Constante	9
	

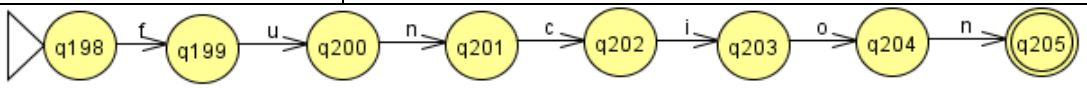
Imprimir en consola	10
	
Inicio de algoritmo	11
	
Fin del algoritmo	12
	
Condición Si	13
	
instrucción entonces	14
	
fin condición si	15
	
Instrucción sino	16
	
Fin de instrucción sino	17
	
Bucle mientras	20
	

Instrucción hacer	19
	
Fin bucle mientras	18
	
Instrucción hasta	21
	
Bucle para	22
	
Fin bucle para	22
	
Instrucción seleccionar	24
	
Instrucción parar	
	
Caso	25
	
Instrucción defecto	26
	

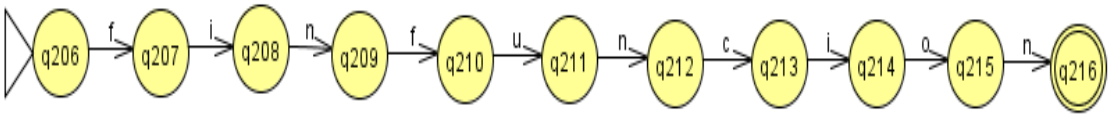
Fin instrucción seleccionar	27



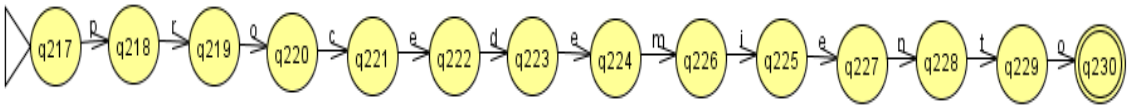
Inicio Función	28



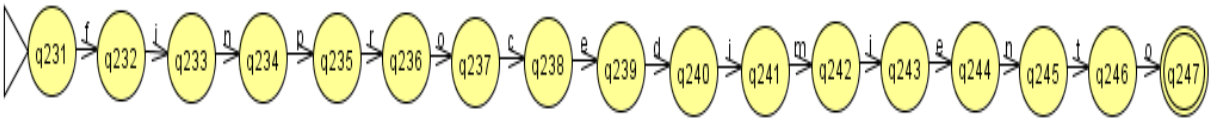
Fin de Función	29



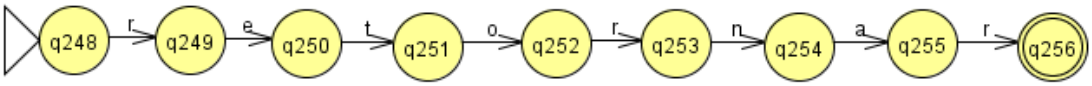
Inicio de Procedimiento	30



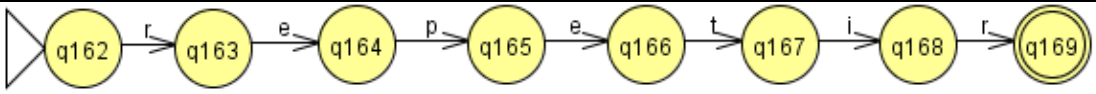
Fin de procedimiento	31



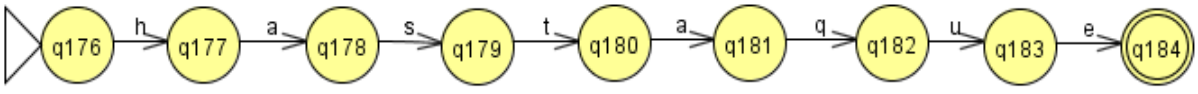
Instrucción retornar	32



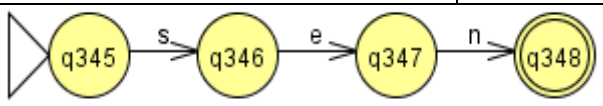
Instrucción repetir	87

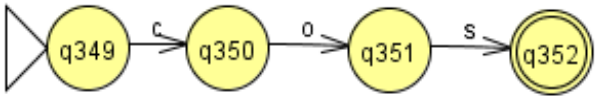
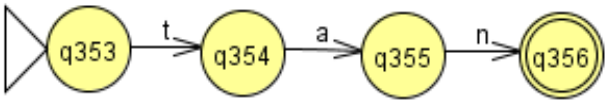
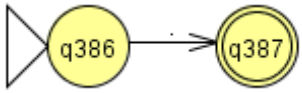
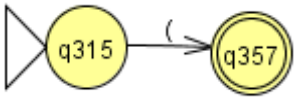
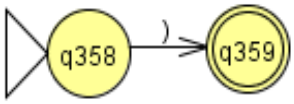
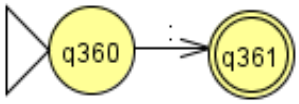
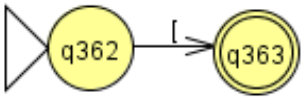
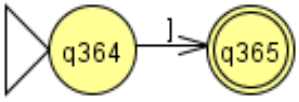


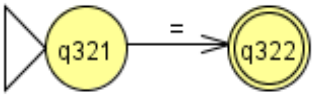
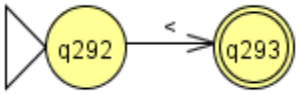
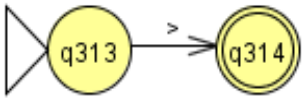
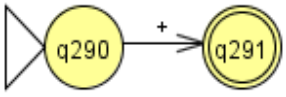
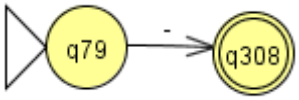
Instrucción hasta que	

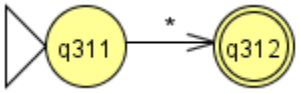
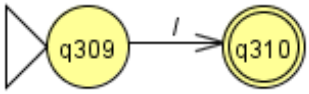
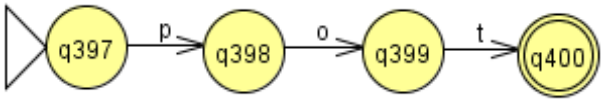
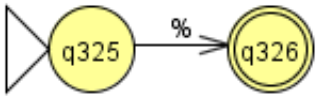
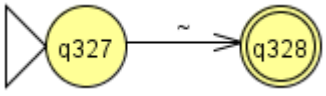


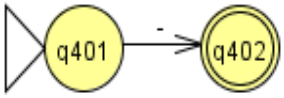
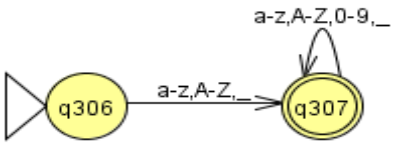
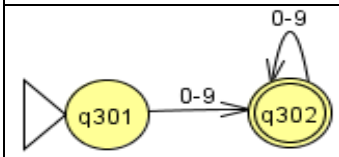
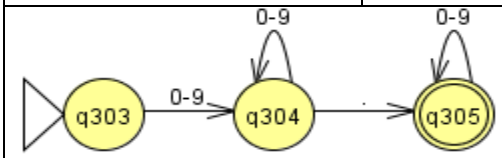
Función Trigonométrica Seno	34



Función Trigonométrica Coseno	33
	
Función Trigonométrica Tangente	35
	
Punto	52
	
Punto y Coma	53
	
Abrir paréntesis	55
	
Cerrar Paréntesis	56
	
Dos puntos	57
	
Abrir corchete	58
	
Cerrar corchete	59
	

Operador Igual	60
	
Operador diferente	61
	
Operador asignación	62
	
Operador menor que	65
	
Operador mayor que	64
	
Operador menor o igual	63
	
Operador mayor o igual	66
	
Operador suma	67
	
Operador resta	68
	

Operador producto	69
	
Operador división	70
	
Operador Potencia	36
	
Operador residuo	71
	
Operador raíz cuadrada	37
	
Operador ó lógico	72
	
Operador y lógico	73
	
Operador negación	74
	
Operador coma	54
	

Operador Mas Mas	75
	
Operador Menos Menos	76
	
Menos Unario	51
	
Identificadores	77
	
Números enteros	84
	
Números reales	79
	

F.1.1.3. Autómata General

Figura 10. Autómata General

F.1.2. Análisis Sintáctico

F.1.2.1. Gramática en ANTLR

La gramática definida en ANTLRWorks contiene en primer lugar la salida que se quiere generar, en este caso los AST, así como también el lenguaje de programación que vamos a utilizar que es Java, luego se definen los lexemas y tokens de nuestra gramática así como las reglas gramaticales, además se generan los AST y se controlan algunos errores semánticos.

```
grammar ArkGrammar;

options {
        output=AST;
        language=Java;
}

tokens{
//tipos básicos
 ENTERO = 'entero';
 REAL = 'real';
 BOOL = 'bool';
 CADENA = 'cadena';
 CARACTER = 'caracter';

 //palabras reservadas
 CONSTANTE='const';
 IMPRIMIR = 'imprimir';
 INICIO = 'INICIO';
 FIN = 'FIN';
 SI = 'si';
 ENTONCES = 'entonces';
 FINSI = 'finsi';
 SINO = 'sino';
 FINSINO='finsino';
 MIENTRAS = 'mientras';
 HACER = 'hacer';
```

```
FINMIENTRAS = 'finmientras';
HASTA = 'hasta';
PARA = 'para';
FINPARA = 'finpara';
SELECCIONAR='seleccionar';
CASO='caso';
DEFECTO='defecto';
FINSELECCIONAR='finseleccionar';
REPETIR= 'repetir';
HASTAQUE='hastaque';
PARAR='parar';
FUNCION = 'funcion';
FINFUNCION='finfuncion';
PROCEDIMIENTO='procedimiento';
FINPROCEDIMIENTO='finprocedimiento';
RETORNAR = 'retornar';

//funciones
COSENO= 'cos';
SENO='sen';
TANGENTE='tan';
POTENCIA='pot';
RAIZ='raiz';

//tokens imaginarios
PROGRAMA;
SUBRUTINA;
DECLARACION;
ASIGNACION;
ARREGLO;
DECLARACIONPARAMETROS;
LISTAPARAMETROS;
DECLARAPARAFIN;
LLAMADA;
LISTAEXPRESIONES;
LISTAINSTRUCCIONES;
LISTASUBRUTINA;
METODO;
MENOSUNARIO;

// Separadores
```

```
PUNTO = '.' ;
PUNTO_COMA =  ';' ;
COMA = ',' ;
PARENTESIS_ABIERTO = '(' ;
PARENTESIS_CERRADO = ')' ;
DOS_PUNTOS= ':';
CORCHETE_ABIERTO='[';
CORCHETE_CERRADO=']';

// operadores
OP_IGUAL = '==' ;
OP_DISTINTO = '!=' ;
OP_ASIG = '=' ;
OP_MENOR = '<' ;
OP_MAYOR = '>' ;
OP_MENOR_IGUAL = '<=' ;
OP_MAYOR_IGUAL = '>=' ;
OP_MAS = '+' ;
OP_MENOS = '-' ;
OP_PRODUCTO = '*' ;
OP_DIVISION = '/' ;
OP_RESIDUO = '%';
OP_OLOGICO = '||';
OP_YLOGICO = '&&';
OP_NEGACION= '~';
OP_MAS_MAS= '++';
OP_MENOS_MENOS= '--';
}

@lexer::header{
        package ark.antlr.modelo;
        import ark.mensaje.ReportarError;
        import ark.mensaje.Traducir;
}

@header{
        package ark.antlr.modelo;
        import java.util.HashMap;
        import java.util.Vector;
        import java.util.StringTokenizer;
        import ark.mensaje.ReportarError;
```

```
	import ark.mensaje.Traducir;
}

@lexer::members{

	public ArkGrammarLexer(String s) {
		super(new ANTLRStringStream(s),new RecognizerSharedState());
	}
	public int numErrores=0;
	public String getErrorMessage(RecognitionException e, String []
tokenNames){
		numErrores++;
		String msg= new
Traducir().mensajeEnEspanol(super.getErrorMessage(e, tokenNames));
		new ReportarError("Error Lexico: "+"linea
"+e.line+":"+e.charPositionInLine+" "+msg);
		return msg;
	}
}

@members{
	public HashMap<String, Simbolo> tablaPrograma = new HashMap<String,
Simbolo>();
	public HashMap<String, Simbolo> tablaMetodos  = new HashMap<String,
Simbolo>();
	public HashMap<String,String> tablaTipoParams = new
HashMap<String,String>();
	public Vector<String> vectorConstantes  = new Vector<String>();
	public Vector<String> vectorVarAsig  = new Vector<String>();
	public Vector <String> vectorArreglos = new Vector <String>();
	public Vector<String> vectorSimbolos = null;
	int numVars=0;
	public int numMetodos=0;
	public int sem =0;
	public int numErrores=0;


	public String getErrorMessage(RecognitionException e, String []
tokenNames){
		numErrores++;
		String msg= new
```

```
Traducir().mensajeEnEspanol(super.getErrorMessage(e, tokenNames));
                    new ReportarError("Error Sintactico: "+"linea
"+e.line+":"+e.charPositionInLine+" "+msg);
                    return msg;
          }

}

programa:      principal lista_subrutina
{if (!(input.LA(1) == EOF)) {
          if (numErrores == 0) {
          sem++;
          Token t = (Token) input.LT(1)
          new ReportarError("Error Sintactico: " + "linea
"+t.getLine()+":"+t.getCharPositionInLine()+" Token inesperado, elimine este token:
"+"'"+t.getText()+"'");
          }
}
}-> ^(PROGRAMA principal lista_subrutina);

principal:       'INICIO'^  lista_instrucciones 'FIN'!;

lista_subrutina
          :        (lista_s+=subrutina)* ->^(LISTASUBRUTINA $lista_s*);

subrutina
          :        funcion |procedimiento;

lista_instrucciones : (lista_m+=instruccion)*          -> ^(LISTAINSTRUCCIONES
$lista_m*);

instruccion     :
          inst_declaracion|inst_asignacion|inst_si|inst_para|inst_mientras|imprimir|i
nst_seleccionar|inst_llamada|inst_repetir|inst_masmenos;

funcion          @init{
          vectorSimbolos = new Vector<String>();
}:
'funcion' tipo IDENT '(' lista_declaraciones ')' lista_instrucciones inst_retornar
'finfuncion'
                    {
```

```
        try{
                if(!tablaMetodos.containsKey($IDENT.text)){
                tablaMetodos.put($IDENT.text, new
Simbolo($tipo.text,numMetodos++));
                String str = $lista_declaraciones.text.replaceAll("," , "
").replaceAll("entero", " entero ").replaceAll("real", " real ").replaceAll("cadena", "
cadena ").replaceAll("caracter", " caracter ").replaceAll("bool", " bool ");
                StringTokenizer stk = new StringTokenizer(str);
                String dev="";
                while(stk.hasMoreTokens()){
                String s=stk.nextToken();

        if(s.equals("entero")|s.equals("real")|s.equals("cadena")|s.equals("caracter")
|s.equals("bool")){
                        dev=dev+s+" ";
                }
                }
                tablaTipoParams.put($IDENT.text,dev);
         }else{
              if(numErrores==0){
              sem++;
              Token t =(Token)retval.getStart();
              new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" funcion '"+$IDENT.text+"'  ya existe");
              }
            }
        }catch(Exception e){

        }
         }->^(METODO tipo IDENT lista_instrucciones inst_retornar
lista_declaraciones);

procedimiento @init{
        vectorSimbolos = new Vector<String>();
}:
'procedimiento' IDENT '(' lista_declaraciones ')' lista_instrucciones 'finprocedimiento'
        {
          try{
             if(!tablaMetodos.containsKey($IDENT.text)){
                tablaMetodos.put($IDENT.text, new
Simbolo("vacio",numMetodos++));
```

```
                String str = $lista_declaraciones.text.replaceAll("," , "
").replaceAll("entero", " entero ").replaceAll("real", " real ").replaceAll("cadena", "
cadena ").replaceAll("caracter", " caracter ").replaceAll("bool", " bool ");
                StringTokenizer stk = new StringTokenizer(str);
                String dev="";
                while(stk.hasMoreTokens()){
                String s=stk.nextToken();

        if(s.equals("entero")|s.equals("real")|s.equals("cadena")|s.equals("caracter")
|s.equals("bool")){
                        dev=dev+s+" ";
                }
              }
        tablaTipoParams.put($IDENT.text,dev);
        }else{
              if(numErrores==0){
              sem++;
              Token t =(Token)retval.getStart();
              new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" procedimiento '"+$IDENT.text+"'  ya
existe");
               }
         }
       }catch(Exception e){

       }
         }->^(METODO IDENT lista_instrucciones lista_declaraciones);

funciones
        :       coseno|seno|tangente|raiz|potencia;

coseno:         'cos'^ '('! expresion ')'! ;
seno    :       'sen'^ '('! expresion ')'! ;
tangente:       'tan'^ '('! expresion ')'! ;
raiz    : 'raiz'^ '('! expresion ')'!;
potencia    :   'pot'^ '('! expresion ','! expresion ')'! ;

inst_masmenos
        :       IDENT^ '++' ';'!
        {
        if (vectorSimbolos != null) {
```

```
                if (vectorSimbolos.contains($IDENT.text)) {
                        if(vectorConstantes.contains($IDENT.text)){
                                if(numErrores==0){
                                        sem++;
                                        Token t =(Token)retval.getStart();
                                        new ReportarError("Error Semántico:
"+"línea "+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"'
es una constante no puede ser asignada más de una vez");
                                }
                        }else  if(vectorArreglos.contains($IDENT.text)){
                                if(numErrores==0){
                                        sem++;
                                        Token t =(Token)retval.getStart();
                                        new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' es un
arreglo, su asignación seria: IDENT [numero] ");
                                }
                        }else if(!vectorVarAsig.contains($IDENT.text)){
                                if(numErrores==0){
                                        sem++;
                                        Token t =(Token)retval.getStart();
                                        new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' no ha
sido inicializado");
                                }
                        }
                }else {
                                if (numErrores == 0) {
                                        sem++;
                                        Token t = (Token) retval.getStart();
                                        new ReportarError("Error Semántico: "+"línea "+
t.getLine()+ ": Identificador '"+$IDENT.text+ "'  no ha sido declarado en la presente
rutina");
                                }
                }
        }else{
                        if(tablaPrograma.containsKey($IDENT.text)){
                                if(vectorConstantes.contains($IDENT.text)){
                                        if(numErrores==0){
                                                sem++;
                                                Token t =(Token)retval.getStart();
```

```
                                                    new ReportarError("Error Semántico:
"+"línea "+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"'
es una constante no puede ser asignada más de una vez");
                                                    }
                                            }else  if(vectorArreglos.contains($IDENT.text)){
                                            if(numErrores==0){
                                                    sem++;
                                                    Token t =(Token)retval.getStart();
                                                    new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' es un
arreglo, su asignación seria: IDENT [numero] ");
                                            }
                                    }else if(!vectorVarAsig.contains($IDENT.text)){
                                            if(numErrores==0){
                                                    sem++;
                                                    Token t =(Token)retval.getStart();
                                                    new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' no ha
sido inicializado");
                                            }
                                    }
            }else{
                            if(numErrores==0){
                                    sem++;
                                    Token t =(Token)retval.getStart();
                                    new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' no ha
sido declarado");
                            }
            }
              }
              }
              |IDENT^ '--' ';'!
              {
              if (vectorSimbolos != null) {
                            if (vectorSimbolos.contains($IDENT.text)) {
                                    if(vectorConstantes.contains($IDENT.text)){
                                                    if(numErrores==0){
                                                            sem++;
                                                            Token t =(Token)retval.getStart();
                                                            new ReportarError("Error Semántico:
```

```
"+"línea "+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"' es una constante no puede ser asignada más de una vez");
                                        }
                                }else  if(vectorArreglos.contains($IDENT.text)){
                                        if(numErrores==0){
                                                sem++;
                                                Token t =(Token)retval.getStart();
                                                new ReportarError("Error Semántico: "+"línea "+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' es un arreglo, su asignación seria: IDENT [numero] ");
                                        }
                                }else if(!vectorVarAsig.contains($IDENT.text)){
                                        if(numErrores==0){
                                                sem++;
                                                Token t =(Token)retval.getStart();
                                                new ReportarError("Error Semántico: "+"línea "+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' no ha sido inicializado");
                                        }
                                }
                        }else {
                                        if (numErrores == 0) {
                                                sem++;
                                                Token t = (Token) retval.getStart();
                                                new ReportarError("Error Semántico: "+"línea "+ t.getLine()+ ": Identificador '"+$IDENT.text+ "'  no ha sido declarado en la presente rutina");
                                        }
                        }
                }else{
                                if(tablaPrograma.containsKey($IDENT.text)){
                                        if(vectorConstantes.contains($IDENT.text)){
                                                if(numErrores==0){
                                                        sem++;
                                                        Token t =(Token)retval.getStart();
                                                        new ReportarError("Error Semántico: "+"línea "+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"' es una constante no puede ser asignada más de una vez");
                                                }
                                        }else  if(vectorArreglos.contains($IDENT.text)){
                                        if(numErrores==0){
```

```
                                        sem++;
                                        Token t =(Token)retval.getStart();
                                        new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' es un
arreglo, su asignación seria: IDENT [numero] ");
                                }
                        }else if(!vectorVarAsig.contains($IDENT.text)){
                                if(numErrores==0){
                                        sem++;
                                        Token t =(Token)retval.getStart();
                                        new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' no ha
sido inicializado");
                                }
                        }
        }else{
                if(numErrores==0){
                        sem++;
                        Token t =(Token)retval.getStart();
                        new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' no ha
sido declarado");
                }
        }
          }
          }
          ;
masmenos_para
          :         IDENT^ '++'
          {
          if (vectorSimbolos != null) {
                if (vectorSimbolos.contains($IDENT.text)) {
                        if(vectorConstantes.contains($IDENT.text)){
                                        if(numErrores==0){
                                                sem++;
                                                Token t =(Token)retval.getStart();
                                                new ReportarError("Error Semántico:
"+"línea "+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"'
es una constante no puede ser asignada más de una vez");
                                        }
                        }else  if(vectorArreglos.contains($IDENT.text)){
```

```
                                if(numErrores==0){
                                        sem++;
                                        Token t =(Token)retval.getStart();
                                        new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' es un
arreglo, su asignación seria: IDENT [numero] ");
                                }
                        }else if(!vectorVarAsig.contains($IDENT.text)){
                                if(numErrores==0){
                                        sem++;
                                        Token t =(Token)retval.getStart();
                                        new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' no ha
sido inicializado");
                                }
                        }
                    }else {
                                if (numErrores == 0) {
                                        sem++;
                                        Token t = (Token) retval.getStart();
                                        new ReportarError("Error Semántico: "+"línea "+
t.getLine()+ ": Identificador '"+$IDENT.text+ "'  no ha sido declarado en la presente
rutina");
                                }
                    }
                }else{
                        if(tablaPrograma.containsKey($IDENT.text)){
                                if(vectorConstantes.contains($IDENT.text)){
                                        if(numErrores==0){
                                                sem++;
                                                Token t =(Token)retval.getStart();
                                                new ReportarError("Error Semántico:
"+"línea "+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"'
es una constante no puede ser asignada más de una vez");
                                        }
                                }else  if(vectorArreglos.contains($IDENT.text)){
                                if(numErrores==0){
                                        sem++;
                                        Token t =(Token)retval.getStart();
                                        new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' es un
```

```
arreglo, su asignación seria: IDENT [numero] ");
                                }
                        }else if(!vectorVarAsig.contains($IDENT.text)){
                                if(numErrores==0){
                                        sem++;
                                        Token t =(Token)retval.getStart();
                                        new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' no ha
sido inicializado");
                                }
                        }
        }else{
                if(numErrores==0){
                        sem++;
                        Token t =(Token)retval.getStart();
                        new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' no ha
sido declarado");
                }
        }
          }
          }
          |IDENT^ '--'
          {
          if (vectorSimbolos != null) {
                if (vectorSimbolos.contains($IDENT.text)) {
                        if(vectorConstantes.contains($IDENT.text)){
                                if(numErrores==0){
                                        sem++;
                                        Token t =(Token)retval.getStart();
                                        new ReportarError("Error Semántico:
"+"línea "+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"'
es una constante no puede ser asignada más de una vez");
                                }
                        }else  if(vectorArreglos.contains($IDENT.text)){
                                if(numErrores==0){
                                        sem++;
                                        Token t =(Token)retval.getStart();
                                        new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' es un
arreglo, su asignación seria: IDENT [numero] ");
```

```
                                }
                        }else if(!vectorVarAsig.contains($IDENT.text)){
                                if(numErrores==0){
                                        sem++;
                                        Token t =(Token)retval.getStart();
                                        new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' no ha
sido inicializado");
                                }
                        }
                    }else {
                                if (numErrores == 0) {
                                        sem++;
                                        Token t = (Token) retval.getStart();
                                        new ReportarError("Error Semántico: "+"línea "+
t.getLine()+ ": Identificador '"+$IDENT.text+ "'  no ha sido declarado en la presente
rutina");
                                }
                    }
                }else{
                        if(tablaPrograma.containsKey($IDENT.text)){
                                if(vectorConstantes.contains($IDENT.text)){
                                        if(numErrores==0){
                                                sem++;
                                                Token t =(Token)retval.getStart();
                                                new ReportarError("Error Semántico:
"+"línea "+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"'
es una constante no puede ser asignada más de una vez");
                                        }
                                }else  if(vectorArreglos.contains($IDENT.text)){
                                if(numErrores==0){
                                        sem++;
                                        Token t =(Token)retval.getStart();
                                        new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' es un
arreglo, su asignación seria: IDENT [numero] ");
                                }
                        }else if(!vectorVarAsig.contains($IDENT.text)){
                                if(numErrores==0){
                                        sem++;
                                        Token t =(Token)retval.getStart();
```

```
                                        new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' no ha
sido inicializado");
                                }
                        }
        }else{
                if(numErrores==0){
                        sem++;
                        Token t =(Token)retval.getStart();
                        new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' no ha
sido declarado");
                }
        }
          }
          }
          ;

decl_parafin : IDENT '=' expresion
{
if (vectorSimbolos != null) {
          if (vectorSimbolos.contains($IDENT.text)) {
                        if(vectorConstantes.contains($IDENT.text)){
                        if(numErrores==0){
                                sem++;
                                Token t =(Token)retval.getStart();
                                new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"'  es una
constante no puede ser asignada más de una vez");
                        }
                }else if(vectorArreglos.contains($IDENT.text)){
                        if(numErrores==0){
                                sem++;
                                Token t =(Token)retval.getStart();
                                new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"'  es un
arreglo su sintaxis es IDENT [numero]");
                        }
                }else{

                                vectorVarAsig.add($IDENT.text);
```

```
                }
        }else {
                if (numErrores == 0) {
                        sem++;
                        Token t = (Token) retval.getStart();
                        new ReportarError("Error Semántico: "+"línea "+
t.getLine()+":"+t.getCharPositionInLine()+ " Identificador '"+$IDENT.text+ "' no ha
sido declarado en la presente rutina");
                }
        }
}else{
        if(tablaPrograma.containsKey($IDENT.text)){
                        if(vectorConstantes.contains($IDENT.text)){
                        if(numErrores==0){
                                sem++;
                                Token t =(Token)retval.getStart();
                                new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"'  es una
constante no puede ser asignada más de una vez");
                        }
                }else if(vectorArreglos.contains($IDENT.text)){
                        if(numErrores==0){
                                sem++;
                                Token t =(Token)retval.getStart();
                                new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"'  es un
arreglo su sintaxis es IDENT [numero]");
                        }
                }else{

                                vectorVarAsig.add($IDENT.text);
                }
                }else {
                        if (numErrores == 0) {
                                sem++;
                                Token t = (Token) retval.getStart();
                                new ReportarError("Error Semántico: "+"línea "+
t.getLine()+":"+t.getCharPositionInLine()+ " Identificador '"+$IDENT.text+ "' no ha
sido declarado");
                        }
                }
```

```
}

}->^(DECLARAPARAFIN IDENT expresion)|masmenos_para;

inst_declaracion : tipo IDENT ';'
        {if(!tablaPrograma.containsKey($IDENT.text)){
         tablaPrograma.put($IDENT.text, new Simbolo($tipo.text, numVars++));
         if (vectorSimbolos != null) {
                        vectorSimbolos.add($IDENT.text);
                }
        }else{
             if(numErrores==0){
             sem++;
             Token t =(Token)retval.getStart();
             new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"'  ya
existe");
             }
           }
        } -> ^(DECLARACION tipo IDENT)
        ;

lista_declaraciones
        :       ld+=declaraciones (',' ld+=declaraciones)* -> ^(LISTAPARAMETROS
$ld*)
        |-> ^(LISTAPARAMETROS);

declaraciones :         tipo IDENT
        {if(!tablaPrograma.containsKey($IDENT.text)){
         tablaPrograma.put($IDENT.text, new Simbolo($tipo.text, numVars++));
         vectorVarAsig.add($IDENT.text);
         if (vectorSimbolos != null) {
                        vectorSimbolos.add($IDENT.text);
                }
        }else{
             if(numErrores==0){
             sem++;
             Token t =(Token)retval.getStart();
             new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador de parámetro
'"+$IDENT.text+"'  ya existe");
```

```
            }
          }
        }
 -> ^(DECLARACIONPARAMETROS tipo IDENT);

inst_asignacion : IDENT '=' expresion ';'
{
if (vectorSimbolos != null) {
        if (vectorSimbolos.contains($IDENT.text)) {
                        if(vectorConstantes.contains($IDENT.text)){
                        if(numErrores==0){
                                sem++;
                                Token t =(Token)retval.getStart();
                                new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"'  es una
constante no puede ser asignada más de una vez");
                        }
                }else if(vectorArreglos.contains($IDENT.text)){
                        if(numErrores==0){
                                sem++;
                                Token t =(Token)retval.getStart();
                                new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"'  es un
arreglo su sintaxis es IDENT [numero]");
                        }
                }else{

                                vectorVarAsig.add($IDENT.text);
                }
         }else {
                if (numErrores == 0) {
                        sem++;
                        Token t = (Token) retval.getStart();
                        new ReportarError("Error Semántico: "+"línea "+
t.getLine()+":"+t.getCharPositionInLine()+ " Identificador '"+$IDENT.text+ "' no ha
sido declarado en la presente rutina");
                }
        }
}else{
        if(tablaPrograma.containsKey($IDENT.text)){
                        if(vectorConstantes.contains($IDENT.text)){
```

```
                        if(numErrores==0){
                                sem++;
                                Token t =(Token)retval.getStart();
                                new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"'  es una
constante no puede ser asignada más de una vez");
                        }
                }else if(vectorArreglos.contains($IDENT.text)){
                        if(numErrores==0){
                                sem++;
                                Token t =(Token)retval.getStart();
                                new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"'  es un
arreglo su sintaxis es IDENT [numero]");
                        }
                }else{

                                vectorVarAsig.add($IDENT.text);
                }
                }else {
                        if (numErrores == 0) {
                                sem++;
                                Token t = (Token) retval.getStart();
                                new ReportarError("Error Semántico: "+"línea "+
t.getLine()+":"+t.getCharPositionInLine()+ " Identificador '"+$IDENT.text+ "' no ha
sido declarado");
                        }
        }
}
}
        -> ^(ASIGNACION IDENT expresion)
| tipo IDENT '=' expresion ';'
{if(!tablaPrograma.containsKey($IDENT.text)){
         tablaPrograma.put($IDENT.text, new Simbolo($tipo.text, numVars++));
         vectorVarAsig.add($IDENT.text);
         if (vectorSimbolos != null) {
                        vectorSimbolos.add($IDENT.text);
                 }
        }else{
             if(numErrores==0){
             sem++;
```

```
                Token t =(Token)retval.getStart();
                new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"'  ya
existe");
                }
             }
          }-> ^(ASIGNACION tipo IDENT expresion)| 'const' tipo IDENT '=' expresion ';'
          {if(!tablaPrograma.containsKey($IDENT.text)){
           tablaPrograma.put($IDENT.text, new Simbolo($tipo.text, numVars++));
           vectorConstantes.add($IDENT.text);
           vectorVarAsig.add($IDENT.text);
           if (vectorSimbolos != null) {
                           vectorSimbolos.add($IDENT.text);
                     }
          }else{
                if(numErrores==0){
                sem++;
                Token t =(Token)retval.getStart();
                new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"'  ya
existe");
                }
             }
          }-> ^(ASIGNACION tipo IDENT expresion)
          | tipo '[' ']' IDENT '=' '['expresion']' ';'
          {if(!tablaPrograma.containsKey($IDENT.text)){
           tablaPrograma.put($IDENT.text, new Simbolo($tipo.text, numVars++));
           vectorArreglos.add($IDENT.text);
           if (vectorSimbolos != null) {
                           vectorSimbolos.add($IDENT.text);
                     }
          }else{
                if(numErrores==0){
                sem++;
                Token t =(Token)retval.getStart();
                new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+$IDENT.text+"'  ya
existe");
                }
             }
          }-> ^(ARREGLO tipo IDENT expresion)
```

```
        | IDENT '['expresion']' '=' expresion';'
        {
        if (vectorSimbolos != null) {
                        if (vectorSimbolos.contains($IDENT.text)) {
                                if(!(vectorArreglos.contains($IDENT.text))){
                                if(numErrores==0){
                                        sem++;
                                        Token t =(Token)retval.getStart();
                                        new ReportarError("Error Semántico:
"+"línea "+t.getLine()+":"+t.getCharPositionInLine()+" Arreglo '"+$IDENT.text+"'  no
ha sido declarado");
                                }
                                  }
                        }else {
                                if (numErrores == 0) {
                                        sem++;
                                        Token t = (Token) retval.getStart();
                                        new ReportarError("Error Semántico:
"+"línea "+ t.getLine()+":"+t.getCharPositionInLine()+ " Arreglo '"+$IDENT.text+ "' no
ha sido declarado en la presente rutina");
                                }
                        }
                }else{
                        if(!(vectorArreglos.contains($IDENT.text))){
                                if(numErrores==0){
                                        sem++;
                                        Token t =(Token)retval.getStart();
                                        new ReportarError("Error Semántico:
"+"línea "+t.getLine()+":"+t.getCharPositionInLine()+" Arreglo '"+$IDENT.text+"'  no
ha sido declarado");
                                }
                        }
          }
        }
        -> ^(ARREGLO IDENT expresion expresion)
        ;



imprimir : 'imprimir'^ '('!expresion')'! ';'!;
```

```
inst_si : 'si'^ '('! expresion ')'! 'entonces'! lista_instrucciones 'finsi'! sino_otras ;

sino_otras : 'sino'^ lista_instrucciones 'finsino'!| -> ^(LISTAINSTRUCCIONES);

inst_mientras : 'mientras'^ '('! expresion ')'! 'hacer'! lista_instrucciones (inst_parar)?
'finmientras'! ;

inst_para: 'para'^ '('! inst_asignacion 'hasta'! expresion ';'! decl_parafin ')'! 'hacer'!
lista_instrucciones (inst_parar)? 'finpara'!;

inst_seleccionar
        :       'seleccionar'^ '('! expresion ')'! casos*  defecto 'finseleccionar'! ;
casos   :       'caso'^ (LIT_ENTERO) ':'! lista_instrucciones (inst_parar)?

;

defecto         :       'defecto'^ ':'! lista_instrucciones (inst_parar)? |->
^(LISTAINSTRUCCIONES);

inst_repetir
        :       'repetir'^ lista_instrucciones (inst_parar)? 'hastaque'! '('! expresion
')'!;

inst_retornar : 'retornar'^ expresion ';'!;

inst_parar
        : 'parar'! ';'!;

expresion : expOr ;
expOr : expAnd ('||'^ expAnd)* ;
expAnd : expComp ('&&'^ expComp)* ;
expComp : expMasMenos (
('=='^|'!='^|'>'^|'<'^|'>='^|'<='^) expMasMenos)* ;
expMasMenos : expMultDiv (
('+'^|'-'^) expMultDiv)* ;
expMultDiv : expMenos (
('*'^|'/'^|'%'^) expMenos)* ;
expMenos : '-' expNo-> ^(MENOSUNARIO expNo)
| '+'? expNo -> expNo;
expNo : ('~'^)? acceso ;
acceso : IDENT{
```

```
if (vectorSimbolos != null) {
		if (vectorSimbolos.contains($IDENT.text)) {
			if(tablaPrograma.containsKey($IDENT.text)){
			if(vectorArreglos.contains($IDENT.text)){
				if(numErrores==0){
					sem++;
					Token t =(Token)retval.getStart();
					new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' es un
arreglo, su asignación seria: IDENT [numero] ");
				}
			}else if(!vectorVarAsig.contains($IDENT.text)){
				if(numErrores==0){
					sem++;
					Token t =(Token)retval.getStart();
					new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' no ha
sido inicializado");
				}
			}
		}else{
			if(numErrores==0){
			sem++;
			Token t =(Token)retval.getStart();
			new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' no ha
sido declarado");
			}
		}
		}else {
				if (numErrores == 0) {
					sem++;
					Token t = (Token) retval.getStart();
					new ReportarError("Error Semántico: "+"línea "+
t.getLine()+ ": Identificador '"+$IDENT.text+ "'  no ha sido delarado en la presente
rutina");
				}
		}
	}else{
			if(tablaPrograma.containsKey($IDENT.text)){
				if(vectorArreglos.contains($IDENT.text)){
```

```
                                    if(numErrores==0){
                                            sem++;
                                            Token t =(Token)retval.getStart();
                                            new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' es un
arreglo, su asignación seria: IDENT [numero] ");
                                    }
                            }else if(!vectorVarAsig.contains($IDENT.text)){
                                    if(numErrores==0){
                                            sem++;
                                            Token t =(Token)retval.getStart();
                                            new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' no ha
sido inicializado");
                                    }
                            }
        }else{
                    if(numErrores==0){
                            sem++;
                            Token t =(Token)retval.getStart();
                            new ReportarError("Error Semántico: "+"línea
"+t.getLine()+":"+t.getCharPositionInLine()+" Identificador '"+($IDENT.text)+"' no ha
sido declarado");
                    }
        }
          }
                            }
| literal
| llamada
| arreglo
| funciones
| '('! expresion ')'!
;

arreglo:        IDENT '['expresion']'
{if (vectorSimbolos != null) {
                                    if (vectorSimbolos.contains($IDENT.text)) {
                                            if(!(vectorArreglos.contains($IDENT.text))){
                                            if(numErrores==0){
                                                    sem++;
                                                    Token t =(Token)retval.getStart();
```

```
                                                new ReportarError("Error Semántico:
"+"línea "+t.getLine()+":"+t.getCharPositionInLine()+" Arreglo '"+$IDENT.text+"'  no
ha sido declarado");
                                        }
                                    }
                                }else {
                                        if (numErrores == 0) {
                                                sem++;
                                                Token t = (Token) retval.getStart();
                                                new ReportarError("Error Semántico:
"+"línea "+ t.getLine()+":"+t.getCharPositionInLine()+ " Arreglo '"+$IDENT.text+ "' no
ha sido delarado en la presente rutina");
                                        }
                                }
                        }else{
                                if(!(vectorArreglos.contains($IDENT.text))){
                                        if(numErrores==0){
                                                sem++;
                                                Token t =(Token)retval.getStart();
                                                new ReportarError("Error Semántico:
"+"línea "+t.getLine()+":"+t.getCharPositionInLine()+" Arreglo '"+$IDENT.text+"'  no
ha sido declarado");
                                        }
                                }
                        }
                }
}-> ^(ARREGLO IDENT expresion);

llamada : IDENT '(' lista_expr ')' -> ^(LLAMADA IDENT lista_expr);

inst_llamada
        :       IDENT '(' lista_expr ')' ';' -> ^(LLAMADA IDENT lista_expr);

lista_expr : lista+=expresion (',' lista+=expresion)* -> ^(LISTAEXPRESIONES $lista*)|-
>^(LISTAEXPRESIONES)
        ;


tipo: 'entero'|'real'|'cadena'|'bool'|'caracter';
literal : LIT_ENTERO
| LIT_REAL
| LIT_CADENA
```

```
| LIT_LOGICO
| LIT_CARACTER ;

//Análisis Léxico
fragment VERDADERO:'verdadero';
fragment FALSO:       'falso';
fragment LETRA : 'a'..'z'|'A'..'Z' ;
fragment DIGITO : '0'..'9' ;
LIT_ENTERO : DIGITO+ ;
LIT_REAL : LIT_ENTERO '.' LIT_ENTERO ;
LIT_CADENA : '"' (~('"'|'\n'|'\r'|'\t'))* '"' ;
LIT_CARACTER : '\'' (~('\''|'\n'|'\r'|'\t')) '\'' ;
LIT_LOGICO : VERDADERO|FALSO ;

IDENT : (LETRA|'_')(LETRA|DIGITO|'_')* ;

COMENTARIO : '//' (~('\n'|'\r'))* '\r'? '\n' {$channel=HIDDEN;}
          |          '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;};

WS
  :  (
        ' '
     |   '\r'
     |   '\t'
     |   '\u000C'
     |   '\n'
     )
        {
          skip();
        }
  ;
```

F.1.2.2. Gramática BNF

La gramática se compone de un conjunto de reglas que definen los elementos léxicos y unidades sintácticas, usando notación definida para las gramáticas BNF.

<programa> ::= <principal> <lista_subrutina>

<principal>::= INICIO <lista_instrucciones> FIN

<lista_subrutina> ::= <lista_subrutina'>

<lista_subrutina'> ::= <subrutina> <lista_subrutina'>

<lista_subrutina'> ::= λ

<subrutina> ::= <funcion>

<subrutina> ::= <procedimiento>

<lista_instrucciones> ::= <lista_instrucciones'>

<lista_instrucciones'> ::= <instruccion> <lista_instrucciones'>

<lista_instrucciones'>::= λ

<instruccion> **::=**<inst_declaracion>

<instruccion> **::=**<inst_asignacion>

<instruccion> ::= <inst_si>

<instruccion> **::=** <inst_para>

<instruccion> **::=** <inst_mientras>

<instruccion> **::=** <imprimir>

<instruccion> **::=** <inst_seleccionar>

<instruccion> **::=**<inst_llamada>

<instruccion> **::=**<inst_repetir>

<instruccion> **::=**<inst_masmenos>

<funcion> ::= funcion <tipo> IDENT (<lista_declaraciones>)

<lista_instrucciones> <inst_retornar> finfuncion

<procedimiento> ::= procedimiento IDENT (<lista_declaraciones>) <lista_instrucciones> finprocedimiento

<funciones> ::= <coseno>

<funciones> ::= <seno>

<funciones> ::= <tangente>

<funciones> ::= <raiz>

<funciones> ::= <potencia>

<coseno> ::= cos (<expresion>)

<seno> ::= sen (<expresion>)

<tangente> ::= tan (<expresion>)

<raiz> ::= raiz (<expresion>)

<potencia> ::= pot (<expresion> , <expresion>)

<inst_masmenos> ::= IDENT ++ ;|IDENT -- ;

<masmenos_para> ::= IDENT ++|IDENT --

<decl_parafin> ::= IDENT = <expresion>

<decl_parafin> ::=<masmenos_para>

<inst_declaracion> ::= <tipo> IDENT ;

<lista_declaraciones> ::= <declaraciones> <lista_declaraciones'>

<lista_declaraciones'> ::= , <declaraciones> <lista_declaraciones'>

<lista_declaraciones'> ::= λ

<declaraciones> ::= <tipo> IDENT

<inst_asignacion> ::= IDENT = <expresion> ;

<inst_asignacion> ::= <tipo> IDENT = <expresion> ;

<inst_asignacion> ::= <tipo> [ ] IDENT = [<expresion>];

<inst_asignacion> ::= IDENT [<expresion>] = <expresion>;

<imprimir> ::= imprimir (<expresion>);

<inst_si> ::= si (<expresion>) entonces <lista_instrucciones> finsi

<sino_otras>

<sino_otras> ::= sino <lista_instrucciones> finsino

<sino_otras> ::= λ

<inst_mientras> ::= mientras (<expresion>) hacer <lista_instrucciones> <inst_parar> finmientras

<inst_para> ::= para (<inst_asignacion> hasta <expresion> ; <decl_parafin>) hacer <lista_instrucciones><inst_parar> finpara

<inst_seleccionar> ::= seleccionar (<expresion>) <casos'> <defecto> finseleccionar

<casos'> ::= <casos> <casos'>

<casos'> ::= λ

<casos> ::= caso (LIT_ENTERO) : <lista_instrucciones> <inst_parar>

<defecto> ::= defecto : <lista_instrucciones><inst_parar>

<defecto> ::= λ

<inst_repetir> ::= repetir <lista_instrucciones><inst_parar> hastaque (<expresion>)

<inst_retornar> ::= retornar <expresion> ;

<inst_parar>::= parar;

<inst_parar>::= λ

<expresion> ::= <expOr>

<expOr> ::= <expAnd> <expOr'>

<expOr'> ::= || <expAnd> <expOr'>

<expOr'> ::= λ

<expAnd> ::= <expComp> <expAmd'>

<expand'> ::= && <expComp> <expand'>

<expand'> ::= λ

<expComp> ::= < expMasMenos> <expComp'>

<expComp'>::= == <expMasMenos> <expComp'>

<expComp'>::= != <expMasMenos> <expComp'>

<expComp'>::= > <expMasMenos> <expComp'>

<expComp'>::= < <expMasMenos> <expComp'>

<expComp'>::= >= <expMasMenos> <expComp'>

<expComp'>::= <= <expMasMenos> <expComp'>

<expComp'>::= λ

<expMasMenos> ::= <expMultDiv> <expMasMenos'>

<expMasMenos'>::= + <expMultDiv> <expMasMenos'>

<expMasMenos'>::= - <expMultDiv> <expMasMenos'>

<expMasMenos'>::= λ

<expMultDiv'> ::= <expMenos><expMultDiv'>

<expMultDiv'> ::= * <expMenos><expMultDiv'>

<expMultDiv'> ::= / <expMenos><expMultDiv'>

<expMultDiv'> ::= % <expMenos><expMultDiv'>

<expMultDiv> ::= λ

<expMenos> ::= - <expNo>

<expMenos> ::= <expMenos'> <expNo>

<expMenos'> ::= +

<expMenos'> ::= λ

<expNo> ::= <expNo'> <acceso>

<expNo'> ::= ~

<expNo'> ::= λ

<acceso> ::= IDENT

<acceso> ::= <literal>

<acceso> ::= <llamada>

<acceso> ::= <arreglo>

<acceso> **::=** <funciones>

<acceso> **::=** (<expresion>)

<arreglo> **::=** IDENT [<expresion>]

<llamada> **::=** IDENT (<lista_expr>)

<inst_llamada> **::=** IDENT (<lista_expr>);

<lista_expr> **::=** <expresion> <lista_expr'>

<lista_expr'> **::=** , <expresion> <lista_expr'>

<lista_expr'> **::=** λ

<tipo> **::=** entero

<tipo> **::=**real

<tipo> **::=**cadena

<tipo> **::=**bool

<tipo> **::=**caracter

<literal> **::=** LIT_ENTERO

<literal> **::=** LIT_REAL

<literal> **::=** LIT_CADENA

<literal> **::=** LIT_LOGICO

<literal> **::=** LIT_CARACTER

```
//Analisis Léxico

fragment LETRA : 'a'..'z'|'A'..'Z' ;
fragment DIGITO : '0'..'9' ;
LIT_ENTERO : DIGITO+ ;
LIT_REAL : LIT_ENTERO '.' LIT_ENTERO ;
LIT_CADENA : '"' (~('"'|'\n'|'\r'|'\t'))* '"' ;
LIT_CARACTER : '\'' (~('\''|'\n'|'\r'|'\t')) '\'' ;
```

```
LIT_LOGICO : 'verdadero'|'falso' ;

IDENT : (LETRA|'_')(LETRA|DIGITO|'_')* ;

COMENTARIO : '//' (~('\n'|'\r'))* '\r'? '\n' {$channel=HIDDEN;}
    |        '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;};

WS  :  ( ' ' |  '\r'  |  '\t'  |  '\u000C'  |  '\n'  )
```

F.1.2.3. Primeros

Los primeros sirven para determinar cuáles serán los no terminales que podrían estar al inicio de una instrucción, se definen de la siguiente manera:

Tabla 9. Primeros

NO TERMINALES	PRIMEROS
programa	{INICIO}
principal	{INICIO}
lista_subrutina	{funcion, procedimiento, λ}
lista_subrutina'	{funcion, procedimiento, λ}
subrutina	{funcion, procedimiento}
lista_instrucciones	{entero, real, cadena, bool, caracter, IDENT, const, si, para, mientras, imprimir, seleccionar, repetir, +, -, λ}
lista_instrucciones'	{entero, real, cadena, bool, caracter, IDENT, const, si, para, mientras, imprimir, seleccionar, repetir, +, -, λ}
instrucción	{entero, real, cadena, bool, caracter, IDENT, const, si, para, mientras, imprimir, seleccionar, -, +, repetir}
función	{funcion}
procedimiento	{procedimiento}

funciones	{cos, sen, tan, raíz, pot}
coseno	{cos}
seno	{sen}
tangente	{tan}
raíz	{raíz}
potencia	{pot}
inst_masmenos	{IDENT}
masmenos_para	{IDENT}
decl_parafin	{IDENT}
inst_declaracion	{ entero, real, cadena, bool, caracter}
lista_declaraciones	{ entero, real, cadena, bool, caracter}
lista_declaraciones'	{, λ}
declaraciones	{ entero, real, cadena, bool, carácter, λ}
inst_asignacion	{IDENT, entero, real, cadena, bool, caracter}
imprimir	{imprimir}
inst_si	{si}
sino_otras	{sino}
inst_mientras	{mientras}
inst_para	{para}
inst_seleccionar	{seleccionar}
casos	{caso}
defecto	{defecto, λ}
casos'	{caso, λ}
inst_repetir	{repetir}
inst_retornar	{retornar}
Inst_parar	{parar, λ}
expresión	{-, +}
expOr	{-, +}
expOr'	{||, λ}
expAnd	{-, +}
expAnd'	{&&, λ}

expComp	{-, +}
expComp'	{==, !=, >, <, >=, <=, λ}
expMasMenos	{-, +}
expMasMenos'	{+, -, λ}
expMulDiv	{-, +}
expMulDiv'	{*, /, %, λ}
expMenos	{-, +}
expNo	{~, λ}
expNo'	{~, λ}
acceso	{IDENT, LIT_ENTERO, LIT_REAL, LIT_CADENA, LIT_LOGICO, LIT_CARACTER, (}
arreglo	{IDENT}
llamada	{IDENT}
inst_llamada	{IDENT}
lista_expr	{-, +}
lista_expr'	{, λ}
tipo	{entero, real, cadena, bool, caracter}
literal	{LIT_ENTERO, LIT_REAL, LIT_CADENA, LIT_LOGICO, LIT_CARACTER}

F.1.2.4. Siguientes

Los siguientes son, como su nombre lo indica los no terminales que van a seguir a cualquier término de la gramática, nos sirven para determinar errores sintácticos, se definen de la siguiente manera:

Tabla 10. Siguientes

NO TERMINALES	SIGUIENTES
programa	{$}
principal	{funcion, procedimiento, $}
lista_subrutina	{$}
lista_subrutina'	{$}
subrutina	{funcion, procedimiento}
lista_instrucciones	{FIN, parar}
lista_instrucciones'	{FIN}
instrucción	{entero, real, cadena, bool, caracter, IDENT, si, para, mientras, imprimir, seleccionar, -, +, repetir, finprincipal, retornar, finprocedimiento, finsi, finmientras, finpara, hastaque}
función	{funcion, procedimiento, entero, real, cadena, bool, caracter, IDENT, FIN}
procedimiento	{funcion, procedimiento, entero, real, cadena, bool, caracter, IDENT, FIN}
funciones	{*, /, %, -, +, ==, !=, >, <, >=, <=, $$, ||,), ;, ,}
coseno	{*, /, %, -, +, ==, !=, >, <, >=, <=, $$, ||,), ;, ,}
seno	{*, /, %, -, +, ==, !=, >, <, >=, <=, $$, ||,), ;, ,}
tangente	{*, /, %, -, +, ==, !=, >, <, >=, <=, $$, ||,), ;, ,}
raíz	{*, /, %, -, +, ==, !=, >, <, >=, <=, $$, ||,), ;, ,}
potencia	{*, /, %, -, +, ==, !=, >, <, >=, <=, $$, ||,), ;, ,}
inst_masmenos	{entero, real, cadena, bool, caracter, IDENT, si, para,

	mientras, imprimir, seleccionar, -, +, repetir, finprincipal, retornar, finprocedimiento, finsi, finmientras, finpara, hastaque}
masmenos_para	{IDENT}
decl_parafin	{)}
inst_declaracion	{funcion, procedimiento, entero, real, cadena, bool, caracter, IDENT, FIN, si, para, mientras, imprimir, seleccionar, -, +, repetir, finprincipal, retornar, finprocediemiento, finsi, finmientras, finpara, hastaque}
lista_declaraciones	{)}
lista_declaraciones'	{)}
declaraciones	{, ,)}
inst_asignacion	{funcion, procedimiento, entero, real, cadena, bool, caracter, IDENT, FIN, si, para, mientras, imprimir, seleccionar, -, +, repetir, finprincipal, retornar, finprocediemiento, finsi, finmientras, finpara, hastaque, hasta}
imprimir	{entero, real, cadena, bool, caracter, IDENT, si, para, mientras, imprimir, seleccionar, -, +, repetir, finprincipal, retornar, finprocedimiento, finsi, finmientras, finpara, hastaque}
inst_si	{entero, real, cadena, bool, caracter, IDENT, si, para, mientras, imprimir, seleccionar, -, +, repetir, finprincipal, retornar, finprocedimiento, finsi, finmientras, finpara, hastaque}
sino_otras	{entero, real, cadena, bool, caracter, IDENT, si, para, mientras, imprimir, seleccionar, -, +, repetir, finprincipal, retornar, finprocedimiento, finsi, finmientras, finpara, hastaque}
inst_mientras	{entero, real, cadena, bool, caracter, IDENT, si, para,

	mientras, imprimir, seleccionar, -, +, repetir, finprincipal, retornar, finprocedimiento, finsi, finmientras, finpara, hastaque}
inst_para	{entero, real, cadena, bool, caracter, IDENT, si, para, mientras, imprimir, seleccionar, -, +, repetir, finprincipal, retornar, finprocedimiento, finsi, finmientras, finpara, hastaque}
inst_seleccionar	{entero, real, cadena, bool, caracter, IDENT, si, para, mientras, imprimir, seleccionar, -, +, repetir, finprincipal, retornar, finprocedimiento, finsi, finmientras, finpara, hastaque}
casos	{caso, defecto}
defecto	{finseleccionar}
casos'	{defecto, finseleccionar}
inst_repetir	{entero, real, cadena, bool, caracter, IDENT, si, para, mientras, imprimir, seleccionar, -, +, repetir, finprincipal, retornar, finprocedimiento, finsi, finmientras, finpara, hastaque}
inst_retornar	{finfuncion}
inst_parar	{hastaque, caso, defecto}
expresión	{), ;, ,}
expOr	{), ;, , }
expOr'	{), ;, , }
expAnd	{||,), ;, , }
expAnd'	{||,), ;, , }
expComp	{$$, ||,), ;, , }
expComp'	{$$, ||,), ;, , }
expMasMenos	{==, !=, >, <, >=, <=, $$, ||,), ;, , }
expMasMenos'	{==, !=, >, <, >=, <=, $$, ||,), ;, , }
expMulDiv	{-, +, ==, !=, >, <, >=, <=, $$, ||,), ;, ,}
expMulDiv'	{-, +, ==, !=, >, <, >=, <=, $$, ||,), ;, ,}

expMenos	{*, /, %, -, +, ==, !=, >, <, >=, <=, $$, ||,), ;, ,}
expNo	{*, /, %, -, +, ==, !=, >, <, >=, <=, $$, ||,), ;, ,}
expNo'	{IDENT, LIT_ENTERO, LIT_REAL, LIT_CADENA, LIT_LOGICO, LIT_CARACTER, (}
acceso	{*, /, %, -, +, ==, !=, >, <, >=, <=, $$, ||,), ;, ,}
arreglo	{*, /, %, -, +, ==, !=, >, <, >=, <=, $$, ||,), ;, ,}
llamada	{*, /, %, -, +, ==, !=, >, <, >=, <=, $$, ||,), ;, ,}
inst_llamada	{entero, real, cadena, bool, caracter, IDENT, si, para, mientras, imprimir, seleccionar, -, +, repetir, finprincipal, retornar, finprocedimiento, finsi, finmientras, finpara, hastaque}
lista_expr	{)}
lista_expr'	{)}
tipo	{IDENT}
literal	{*, /, %, -, +, ==, !=, >, <, >=, <=, $$, ||,), ;, ,}

F.1.3. Análisis Semántico

F.1.3.1. Análisis Semántico en ANTLR

En el análisis semántico lo que se hace es extraer los AST desde el análisis sintáctico, para ello se debe eliminar los elementos de la gramática que se encuentran en la parte izquierda del operador ->, conservando los fragmentos de generación de AST, y se recorre los árboles en busca de errores semánticos, para recorrer los arboles se utiliza un análisis descendente recursivo.

```
tree grammar ArkSemantico;
options{

tokenVocab=ArkGrammar;
ASTLabelType=CompAST;
language = Java;

}

@header{
package ark.antlr.modelo;
import java.util.HashMap;
import java.util.Vector;
import java.util.StringTokenizer;
import ark.mensaje.ReportarError;
}

@members{

public HashMap<String,Simbolo> tablaPrograma;
public HashMap<String,Simbolo> tablaMetodos;
public HashMap<String,String> tablaTipoParams;
public Vector <Integer> vectorSelecc;
public int numErrores=0;
public ark.mensaje.ReportarError reportarError;
```

```
private String aux="";
private CompAST identF;

public boolean comprobarTipoExpOrAnd(String t1, String t2){
        boolean res=false;
        if(t1.equals("bool")&&t2.equals("bool")){
                res=true;
        }
        return res;
}

public boolean comprobarTipoExpComp(String op,String t1, String t2){
        boolean res=false;
        for(int i=0;i<op.length();i++){
                if(op.charAt(i)=='<'||op.charAt(i)=='>'){
                res= tipoCompArit(t1,t2);break;
        }
                if(op.charAt(i)=='='||op.charAt(i)=='!'){
                res= tipoCompTodos(t1,t2);break;
        }
        }
        return res;
   }

   public boolean tipoCompArit(String t1, String t2){
        boolean res=false;

        if((t1.equals("real")&&t2.equals("real"))||(t1.equals("entero")&&t2.equals("enter
o"))||(t1.equals("entero")&&t2.equals("real"))||(t1.equals("real")&&t2.equals("entero")))
{
          res=true;
        }
          return res;
        }

   public boolean tipoCompTodos(String t1, String t2){
        boolean res=false;

        if(t1.equals(t2)||(t1.equals("entero")&&t2.equals("real"))||(t1.equals("real")&&t2
```

```
.equals("entero"))){
            res=true;
          }
            return res;
          }

          public boolean comprobarTipoExpArit(String op, String t1, String t2){
          boolean res=false;
          for(int i=0;i<op.length();i++){
                    if(op.charAt(i)=='+'){
                    res= comprobarTipoExpCadena(t1,t2);break;
          }

          }

          if((t1.equals("entero") && t2.equals("entero"))     ||
     (t1.equals("real") && t2.equals("real")) ||
            (t1.equals("entero") && t2.equals("real"))   ||
     (t1.equals("real") && t2.equals("entero"))
     ){
            res=true;
          }
            return res;
          }

          public boolean comprobarTipoExpCadena(String t1, String t2){
          boolean res=false;

          if((t1.equals("cadena") || t2.equals("cadena"))  ||
        (t1.equals("caracter") && t2.equals("entero"))||
        (t1.equals("entero") && t2.equals("caracter"))
        ){
            res=true;
          }
            return res;
          }

          public String tipoExpresion(String t1, String t2){
          String tipo="";
```

```
          if(t1.equals(t2)){
            tipo=t1;
          }else

          if((t1.equals("cadena") || t2.equals("cadena"))){
            tipo="cadena";
          }else
          if((t1.equals("entero") || t2.equals("caracter"))){
            tipo="entero";
          }else
      if((t1.equals("caracter") || t2.equals("entero"))){
        tipo="entero";
       }else
          if((t1.equals("real") || t2.equals("entero"))){
            tipo="real";
          }else
          if((t1.equals("entero") || t2.equals("real"))){
            tipo="real";
          }
          return tipo;
          }

          public boolean comprobarTipoAsig(String t1, String t2){
          boolean res=false;
          if(t1.equals(t2) ||
                              (t1.equals("entero") && t2.equals("real")) ||
                               (t1.equals("caracter") && t2.equals("entero")) ){
            res=true;
          }
            return res;
          }

      public void registrarError(int linea,int posicion,String msg){
                    numErrores++;
                    new   ReportarError("Error   Semántico:   "+"línea   "+línea+":"+posicion+"
"+msg);
          }
}
```

```
programa[HashMap<String, Simbolo> tablaPrograma, HashMap<String, Simbolo> tablaMetodos, HashMap<String, String> tablaTipoParams]

@init{
	this.tablaPrograma=tablaPrograma;
	this.tablaMetodos=tablaMetodos;
	this.tablaTipoParams=tablaTipoParams;
}
:^(PROGRAMA principal lista_subrutina);

principal
	:^('INICIO' lista_instrucciones);

subrutina
	: funcion|procedimiento;

lista_subrutina
	:	^(LISTASUBRUTINA  subrutina*);

instruccion	:
	inst_declaracion|inst_asignacion|inst_si|inst_para|inst_mientras|imprimir|inst_seleccionar|inst_llamada|inst_repetir|inst_masmenos;

lista_instrucciones : ^(LISTAINSTRUCCIONES instruccion*);

funcion
:
^(METODO tipo IDENT  lista_instrucciones inst_retornar lista_declaraciones)
{
identF= $IDENT
identF.simbolo = (Simbolo)tablaMetodos.get($IDENT.text);
}

;

procedimiento:

	^(METODO IDENT lista_instrucciones lista_declaraciones)
	;
```

```
funciones returns [String tipo]
@init {
 $tipo="real";
}
        :       coseno|seno|tangente|raiz|potencia;

coseno :        ^(fun='cos' ex=expresion)
{
        if(!($ex.tipo.equals("entero") || $ex.tipo.equals("real"))){
        registrarError($fun.getLine(), $fun.getCharPositionInLine(), "tipo incorrecto en funcion coseno");
    }
};

seno    :       ^(fun='sen' ex=expresion)
{
        if(!($ex.tipo.equals("entero") || $ex.tipo.equals("real"))){
        registrarError($fun.getLine(), $fun.getCharPositionInLine(), "tipo incorrecto en funcion seno");
    }
};
tangente        :       ^(fun='tan' ex=expresion)
{
        if(!($ex.tipo.equals("entero") || $ex.tipo.equals("real"))){
        registrarError($fun.getLine(), $fun.getCharPositionInLine(), "tipo incorrecto en funcion tangente");
    }
};

raiz    :       ^(fun='raiz' ex=expresion)
{
        if(!($ex.tipo.equals("entero") || $ex.tipo.equals("real"))){
        registrarError($fun.getLine(), $fun.getCharPositionInLine(), "tipo incorrecto en funcion raiz");
    }
};
potencia        :       ^(fun='pot' e1=expresion e2=expresion)
{
```

```
        if(!($e1.tipo.equals("entero") || $e1.tipo.equals("real"))){
        registrarError($fun.getLine(), $fun.getCharPositionInLine(), "tipo incorrecto en
funcion potencia");
    }
    if(!($e2.tipo.equals("entero") || $e2.tipo.equals("real"))){
        registrarError($fun.getLine(), $fun.getCharPositionInLine(), "tipo incorrecto en
funcion potencia");
    }
};

inst_masmenos
@init {
CompAST root = (CompAST)input.LT(1);
}
        :       ^(IDENT '++'){
        if(tablaPrograma.containsKey($IDENT.text)){
                $IDENT.simbolo = (Simbolo)tablaPrograma.get($IDENT.text);
                String s = $IDENT.simbolo.tipo;
                if(!(s.equals("entero")||s.equals("real"))){
                        registrarError(root.getLine(),          root.getCharPositionInLine(),
"Operador ++ solo puede ser asignado a variables enteras o reales");
                }
        }

        }|^(IDENT '--'){
        if(tablaPrograma.containsKey($IDENT.text)){
                $IDENT.simbolo = (Simbolo)tablaPrograma.get($IDENT.text);
                String s = $IDENT.simbolo.tipo;
                if(!(s.equals("entero")||s.equals("real"))){
                        registrarError(root.getLine(),
root.getCharPositionInLine(),"Operador -- solo puede ser asignado a variables enteras o
reales");
                }
        }
        };
masmenos_para
@init {
CompAST root = (CompAST)input.LT(1);
}
```

```
        :       ^(IDENT '++'){
        if(tablaPrograma.containsKey($IDENT.text)){
                $IDENT.simbolo = (Simbolo)tablaPrograma.get($IDENT.text);
                String s = $IDENT.simbolo.tipo;
                if(!(s.equals("entero")||s.equals("real"))){

        registrarError(root.getLine(),root.getCharPositionInLine(),"Operador ++ solo puede
ser asignado a variables enteras o reales");
                }
        }
        }|^(IDENT '--'){
        if(tablaPrograma.containsKey($IDENT.text)){
                $IDENT.simbolo = (Simbolo)tablaPrograma.get($IDENT.text);
                String s = $IDENT.simbolo.tipo;
                if(!(s.equals("entero")||s.equals("real"))){

        registrarError(root.getLine(),root.getCharPositionInLine(),"Operador -- solo puede
ser asignado a variables enteras o reales");
                }
        }
        };

lista_declaraciones
        :       ^(LISTAPARAMETROS declaraciones*)
        ;

declaraciones
:  ^(DECLARACIONPARAMETROS tipo IDENT)
;

inst_declaracion
: ^(DECLARACION tipo IDENT)  ;

inst_asignacion
@init {
CompAST root = (CompAST)input.LT(1);
}
  : ^(ASIGNACION IDENT e1=expresion) {
```

```
        if(tablaPrograma.containsKey($IDENT.text)){
           root.extipo = $e1.tipo;
            $IDENT.simbolo = (Simbolo)tablaPrograma.get($IDENT.text);
            if(!comprobarTipoAsig(root.extipo, $IDENT.simbolo.tipo)){
                registrarError(root.getLine(),root.getCharPositionInLine(),"Tipo incorrecto en
asignacion");
            }
        }
}
           |^(ASIGNACION tipo IDENT e1=expresion){

        if(tablaPrograma.containsKey($IDENT.text)){
            root.extipo = $e1.tipo;
            $IDENT.simbolo = (Simbolo)tablaPrograma.get($IDENT.text);
            if(!comprobarTipoAsig(root.extipo, $IDENT.simbolo.tipo)){
                registrarError(root.getLine(),root.getCharPositionInLine(), "Tipo incorrecto en
asignacion");
            }
        }
}       |^(ARREGLO tipo IDENT e1=expresion){
        if(tablaPrograma.containsKey($IDENT.text)){
           root.extipo = $e1.tipo;
           if(!root.extipo.equals("entero")){
                    registrarError(root.getLine(),root.getCharPositionInLine(), "Tipo incorrecto
en tamaño de arreglo");
           }
        }
}           |^(ARREGLO IDENT e1=expresion e2=expresion){
        if(tablaPrograma.containsKey($IDENT.text)){
           root.extipo = $e1.tipo;
           if(!root.extipo.equals("entero")){
                    registrarError(root.getLine(),root.getCharPositionInLine(), "Tipo incorrecto
en arreglo [entero]");
           }
           root.extipo = $e2.tipo;
            $IDENT.simbolo = (Simbolo)tablaPrograma.get($IDENT.text);
            if(!comprobarTipoAsig(root.extipo, $IDENT.simbolo.tipo)){
                registrarError(root.getLine(),root.getCharPositionInLine(), "Tipo incorrecto en
asignacion de arreglo");
```

```
            }
        }
}
;

inst_si : ^(ins='si' e1=expresion lista_instrucciones sino_otras) {
        if(!$e1.tipo.equals("bool")){
        registrarError($ins.getLine(),$ins.getCharPositionInLine(), "Tipo incorrecto en
instruccion si");
    }
};

sino_otras      :       ^('sino'lista_instrucciones)|LISTAINSTRUCCIONES;

inst_mientras : ^(ins='mientras' e1=expresion lista_instrucciones) {

        if(!$e1.tipo.equals("bool")){
        registrarError($ins.getLine(),$ins.getCharPositionInLine(), "Tipo incorrecto en
instruccion mientras");
    }
};

inst_seleccionar
        :       ^(ins='seleccionar' ex=expresion casos* defecto)
        {
        if(!($ex.tipo.equals("entero"))&& !($ex.tipo.equals("caracter"))){
        registrarError($ins.getLine(),$ins.getCharPositionInLine(), "Tipo incorrecto en
instruccion seleccionar");
    }
};

casos
@init {
        vectorSelecc= new Vector<Integer>();
}
:       ^(ins='caso' le=LIT_ENTERO lista_instrucciones)
{
                if(vectorSelecc.contains(Integer.valueOf($le.text))){
                        registrarError($ins.getLine(),$ins.getCharPositionInLine(), "Caso
```

```
duplicado");
                    }else{
                            vectorSelecc.add(Integer.valueOf($le.text));
                    }
}
;

defecto:        ^('defecto' lista_instrucciones)|LISTAINSTRUCCIONES;

inst_repetir
        :       ^(ins='repetir'  lista_instrucciones ex=expresion)
        {
        if(!$ex.tipo.equals("bool")){
        registrarError($ins.getLine(),$ins.getCharPositionInLine(), "Tipo incorrecto en
instruccion repetir");
    }
};

inst_para : ^(ins='para' inst_asignacion ex=expresion decl_parafin lista_instrucciones) {

        if(!$ex.tipo.equals("bool")){
        registrarError($ins.getLine(),$ins.getCharPositionInLine(), "Tipo incorrecto en
instruccion para");
    }
};

decl_parafin
@init {
CompAST root = (CompAST)input.LT(1);
}
        : ^(DECLARAPARAFIN IDENT e1=expresion){
      if(tablaPrograma.containsKey($IDENT.text)){
        root.extipo = $e1.tipo;
         $IDENT.simbolo = (Simbolo)tablaPrograma.get($IDENT.text);
         if(!comprobarTipoAsig(root.extipo, $IDENT.simbolo.tipo)){
            registrarError(root.getLine(),root.getCharPositionInLine(), "Tipo incorrecto en
asignacion");
         }
      }
```

```
        }|masmenos_para;

imprimir
        :^('imprimir' expresion );

inst_retornar
@init {
CompAST root = (CompAST)input.LT(1);
}
    : ^('retornar' e1=expresion) {
        root.extipo = $e1.tipo;
        if(!comprobarTipoAsig(root.extipo, identF.simbolo.tipo)){
        registrarError(root.getLine(),root.getCharPositionInLine(),"Tipo de retorno no es
igual a funcion declarada "+"'"+identF+"'");
      }

    };

 tipo : 'entero'
    | 'real'
    | 'bool'
    | 'cadena'
    | 'caracter'
    ;

literal returns [String tipo]
@init {
   $tipo="";
   CompAST root = (CompAST)input.LT(1);
}
@after {
   root.extipo = $tipo;
}
   : LIT_ENTERO {$tipo="entero";}
   | LIT_REAL {$tipo="real";}
   | LIT_CADENA {$tipo="cadena";}
   | LIT_LOGICO {$tipo="bool";}
   | LIT_CARACTER {$tipo="caracter";}
   ;
```

```
expresion returns [String tipo]
@init{
$tipo="";
CompAST root = (CompAST) input.LT(1);
}
@after{
root.extipo=$tipo;
}
:
        ^(opOrAnd e1=expresion e2=expresion){
                $tipo="bool";
                if(!comprobarTipoExpOrAnd($e1.tipo, $e2.tipo)){
                        registrarError(root.getLine(),root.getCharPositionInLine(), "Tipos
incorrectos en expresion");
                }

        }
        |^(opc=opComparacion e1=expresion e2=expresion) {
                $tipo="bool";
                root.expSecTipo = $e1.tipo;
                if(!comprobarTipoExpComp($opc.text,$e1.tipo, $e2.tipo)){
        registrarError(root.getLine(),root.getCharPositionInLine(), "Tipos incorrectos en
expresion");
    }
    }

        |^(op=opAritmetico e1=expresion e2=expresion ) {
        $tipo= tipoExpresion($e1.tipo,$e2.tipo);
        if(!comprobarTipoExpArit($op.text,$e1.tipo, $e2.tipo)){
         registrarError(root.getLine(),root.getCharPositionInLine(), "Tipos incorrectos en
expresion");
    }
        }
        |^(MENOSUNARIO e1=expresion) {
      $tipo=$e1.tipo;
          if(!($e1.tipo.equals("entero") || $e1.tipo.equals("real"))){
              registrarError(root.getLine(),root.getCharPositionInLine(), "Tipos incorrectos
```

```
en expresion");
        }
       }
           |^('~' e1=expresion) {
         $tipo=$e1.tipo;
         if(!$e1.tipo.equals("bool")){
            registrarError(root.getLine(),root.getCharPositionInLine(), "Tipos incorrectos en
expression");
         }
       }

           |IDENT {
        if(tablaPrograma.containsKey($IDENT.text)){
          root.simbolo = (Simbolo)tablaPrograma.get($IDENT.text);
             $tipo=root.simbolo.tipo;
           }
       }
           |literal   {$tipo=$literal.tipo;}
           |llamada {$tipo=$llamada.tipo;}
           |arreglo {$tipo=$arreglo.tipo;}
           |funciones {$tipo=$funciones.tipo;}
;

arreglo returns [String tipo]
@init {
$tipo="";
CompAST root = (CompAST)input.LT(1);
}
@after {
root.extipo = $tipo;
}
: ^(ARREGLO IDENT e=expresion) {
      if(tablaPrograma.containsKey($IDENT.text)){
                  root.simbolo = (Simbolo)tablaPrograma.get($IDENT.text);
                  $tipo=root.simbolo.tipo;
         if(!$e.tipo.equals("entero")){
         registrarError(root.getLine(),root.getCharPositionInLine(), "Tipo incorrecto en
arreglo [entero]");
         }
```

```
      }
} ;
llamada returns [String tipo]
@init {
$tipo="";
CompAST root = (CompAST)input.LT(1);
}
@after {
root.extipo = $tipo;
}
: ^(LLAMADA IDENT le=lista_expr) {

      if(tablaMetodos.containsKey($IDENT.text)){
       StringTokenizer str = new StringTokenizer(tablaTipoParams.get($IDENT.text));
        if((str.countTokens())==new Parametro().numComas($lista_expr.text)){
           root.simbolo = (Simbolo)tablaMetodos.get($IDENT.text);
               aux=$IDENT.text;
               $tipo=root.simbolo.tipo;
               }else{
                     registrarError(root.getLine(),root.getCharPositionInLine(),    "funcion    o
procedimiento '" + $IDENT.text + "' no posee el mismo número de parámetros");
               }
           }else{
                     registrarError(root.getLine(),root.getCharPositionInLine(),    "funcion    o
procedimiento '" + $IDENT.text + "' no ha sido declarado");
           }
} ;

inst_llamada
@init {
CompAST root = (CompAST)input.LT(1);
}

: ^(LLAMADA IDENT le=lista_expr) {
    if(tablaMetodos.containsKey($IDENT.text)){
      StringTokenizer str = new StringTokenizer(tablaTipoParams.get($IDENT.text));
       if((str.countTokens())==new Parametro().numComas($lista_expr.text)){
               aux=$IDENT.text;
            }else{
```

```
                    registrarError(root.getLine(), root.getCharPositionInLine(),"funcion o
procedimiento '" + $IDENT.text + "' no posee el mismo número de parámetros");
                }
            }else{
                    registrarError(root.getLine(),root.getCharPositionInLine(), "funcion o
procedimiento '" + $IDENT.text + "' no ha sido declarado");
            }
} ;

lista_expr
@init {
CompAST root = (CompAST)input.LT(1);
}
: ^(LISTAEXPRESIONES exp=expresion*)
{
try{
StringTokenizer str = new StringTokenizer(tablaTipoParams.get(aux));
String s = str.nextToken();
if(!comprobarTipoAsig(exp, s)){
  registrarError(root.getLine(),root.getCharPositionInLine(), "asignacion incorrecta en
llamada '"+aux+"'")
}
}catch(Exception e){
}
};

opAritmetico : '+'|'-'|'*'|'/'|'%' ;

opComparacion : '=='|'!='|'>='|'<='|'>'|'<' ;

opOrAnd          :          '||'|'&&';
```

F.1.3.2. Ejemplos de Árboles Semánticos

A continuación se presentan ejemplos de arboles semánticos que se generan en ANTLRWorks, a partir de la gramática definida.

a. **INICIO**
real b;
entero a =2;
FIN

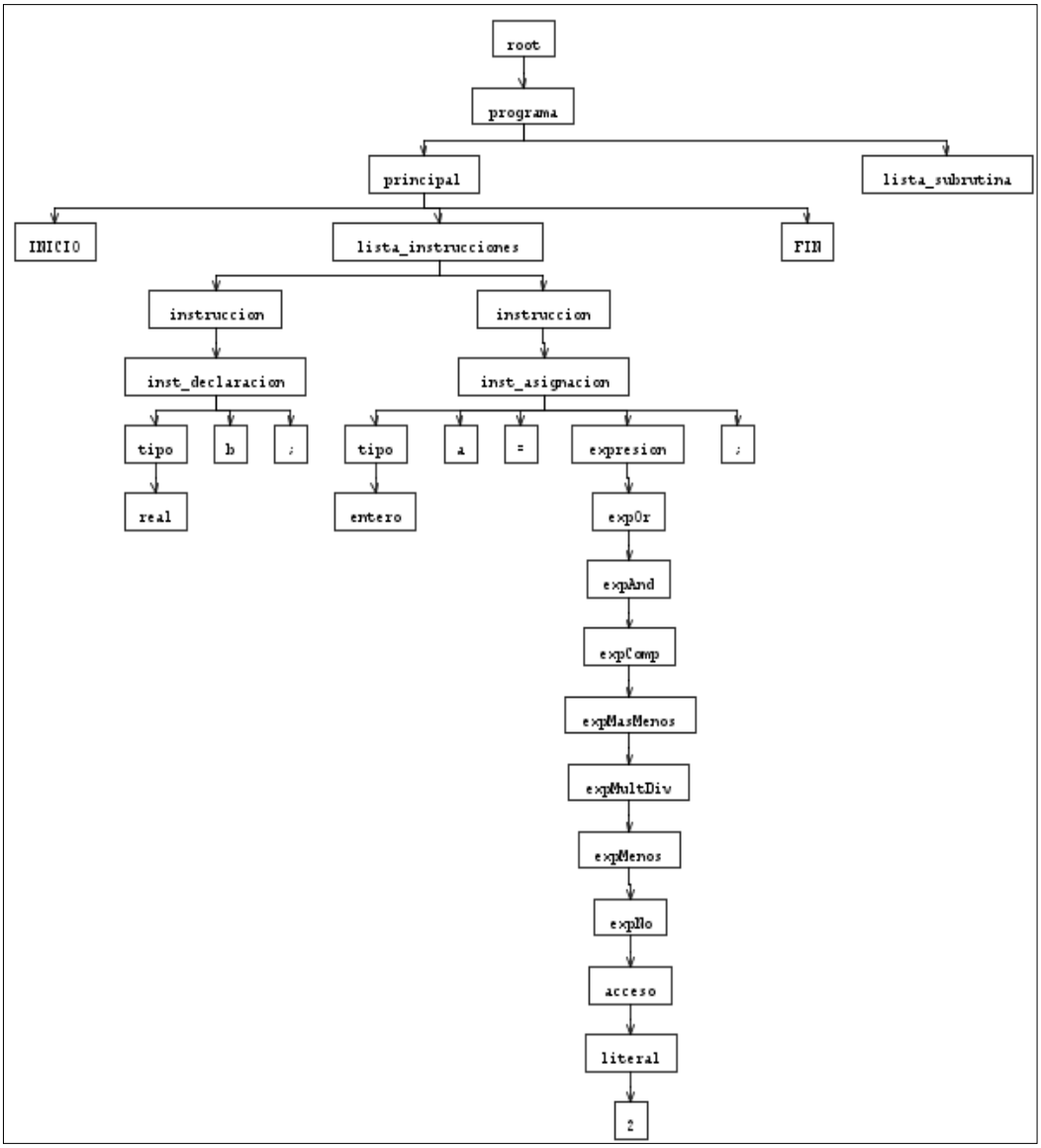


Figura 11. Ejemplo del Árbol Semántico de declaraciones y asignaciones

b.
```
INICIO
FIN
procedimiento nom ()
bool a=falso;
finprocedimiento
```

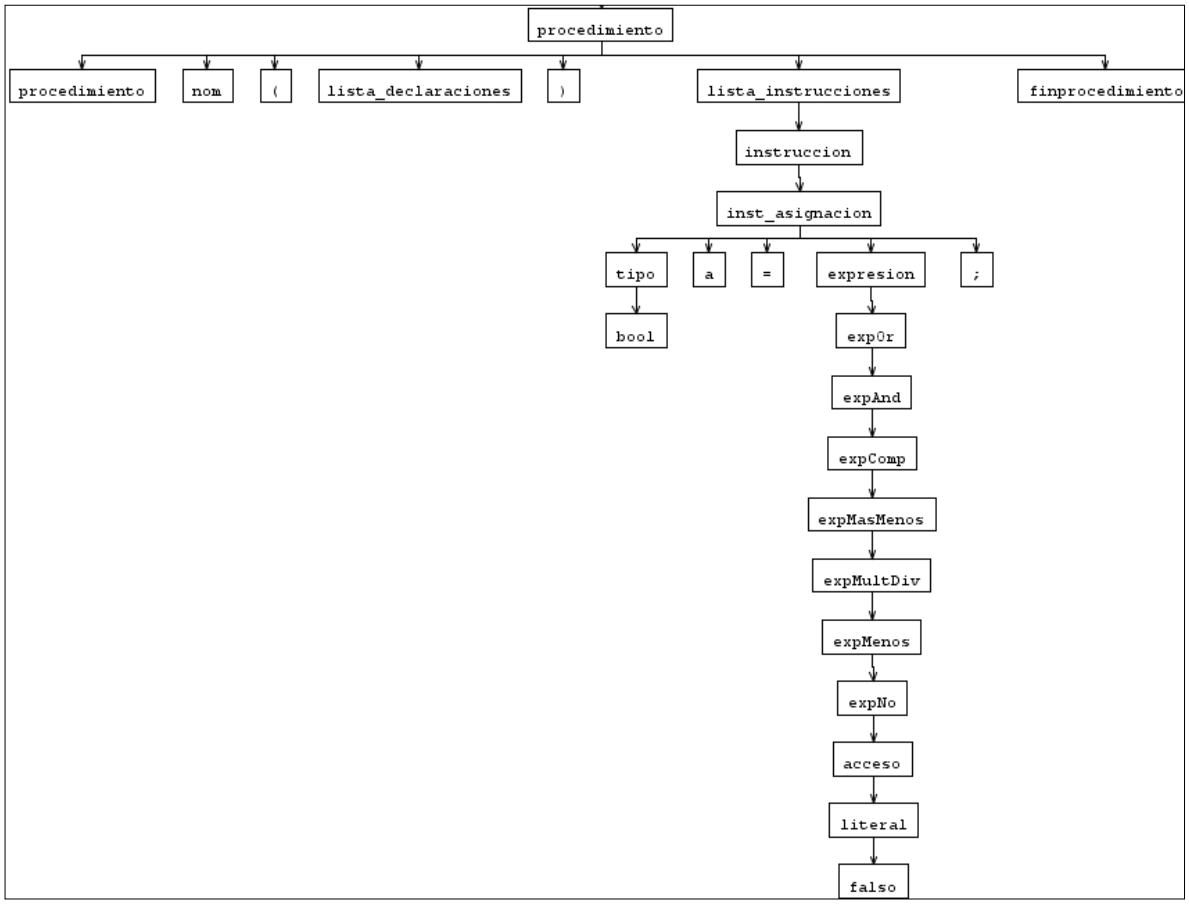


Figura 12. Ejemplo del Árbol Semántico de procedimiento

F.1.4. Generación de Código

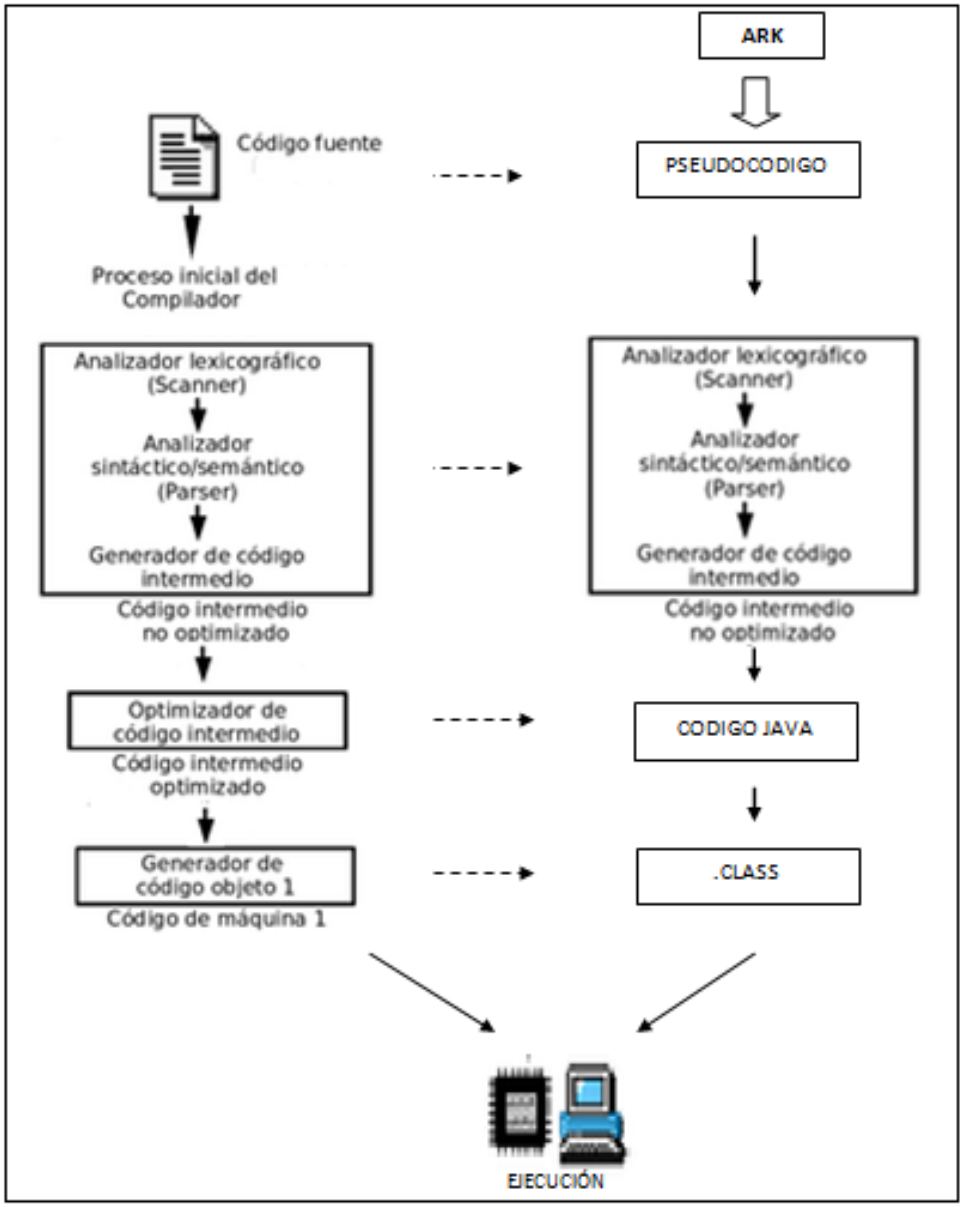


Figura 13. Generación de Código

Una vez realizado el análisis léxico, sintáctico y semántico, es momento de realizar la generación de código que en este caso se lo realiza a través de la conversión o traducción a lenguaje java el mismo que se convierte en el código intermedio del

compilador. Una vez realizada la conversión, se crea un ejecutable de acuerdo al sistema operativo en el que se trabaje, si es Windows se trabaja con un archivo. (bat), si es Linux la extensión es (.sh), de ésta manera se genera el código intermedio para que sea ejecutado en el archivo ejecutable, por lo que por un archivo (.ark), se generan un (.java) (.class) y el ejecutable de acuerdo a Sistema Operativo en el que se trabaje. Para un mejor entendimiento se puede visualizar la figura 12, que ejemplifica el proceso de ejecución de un programa escrito en lenguaje ARK, de esta manera los pasos a seguir son:

- Se tiene el código fuente, el mismo que es el programa escrito en lenguaje ARK.
- Se realiza el análisis léxico al código fuente para detectar las palabras reservadas, separadores y de más lexemas que componen al lenguaje ARK.
- Realizar el análisis sintáctico para la comprobación de las reglas gramaticales establecidas.
- Realizar el análisis semántico, de acuerdo a las reglas semánticas predeterminadas.
- Si la compilación es correcta, se procede a generar código intermedio (Proceso de conversión a lenguaje ARK) por los tanto genera un archivo con la extensión .java.
- Realizar la creación del archivo ejecutable dependiendo del Sistema Operativo en el que se trabaje, para Windows .bat y Linux .sh.
- A través del ejecutable, realizar la compilación del código intermedio y genera el archivo .class que viene a ser el código objeto.
- A través del ejecutable, realizar la ejecución del código objeto es decir del archivo .class.
- Finalmente cerrar el proceso de ejecución.

F.1.5. Pruebas de Validación

Las pruebas son importantes en el proceso de desarrollo de una aplicación puesto que es en esta etapa donde se detectan y corrigen los errores que pudiesen existir, lo que permite asegurar la calidad en el desarrollo.

En este apartado se mostrará el proceso de validación de la aplicación que se debió realizar para comprobar que la aplicación satisface los requerimientos de los usuarios, para lo que se desarrollo un plan de pruebas donde se aplicó una encuesta a dichos usuarios y así constatar el funcionamiento de la aplicación, además se hizo un análisis de los resultados de la encuesta aplicada y por último se muestra un informe final de las pruebas de validación.

F.1.5.1. Herramienta para la Validación

UNIVERSIDAD NACIONAL DE LOJA AREA DE LA ENERGIA,
LAS INDUSTRIAS Y LOS RECURSOS NATURALES NO RENOVABLES
INGENIERIA EN SISTEMAS

1. ¿Considera usted que la aplicación es amigable con el usuario?
   Si ( ) No ( )
   ¿Por qué?.....
   ……

2. ¿Cree usted que el Diseño de la aplicación es conveniente para facilitar las actividades académicas que usted realiza?
   Si ( ) No ( )
   ¿Por qué?.....
   ……

3. ¿Piensa usted que la información presentada en la aplicación es suficiente para el desarrollo de sus actividades académicas?

Si ( ) No ( )
¿Por qué?.....
……

4. ¿Tuvo algún inconveniente al utilizar la aplicación?
   Si ( ) No ( )
   ¿Cuál?.....
   ……

5. ¿Considera razonable el tiempo de ejecución de la aplicación?
   Si ( ) No ( )
   ¿Por qué?.....
   ……

6. ¿Considera que la aplicación es útil en cuanto a la agilización del proceso de corrida de algoritmos y detección de errores?
   Si ( ) No ( )
   ¿Por qué?.....
   ……

7. ¿Cree usted que el almacenamiento de información que ofrece la aplicación es rápido y seguro?
   Si ( ) No ( )
   ¿Por qué?.....
   ……

8. ¿Piensa usted que con la aplicación podrá practicar más y mejorar los conocimientos adquiridos en el aula?
   Si ( ) No ( )
   ¿Por qué?.....
   ……

GRACIAS POR SU COLABORACIÓN

F.1.5.2. Análisis de Resultados de Validación

1. **¿Considera usted que la aplicación es amigable con el usuario?**

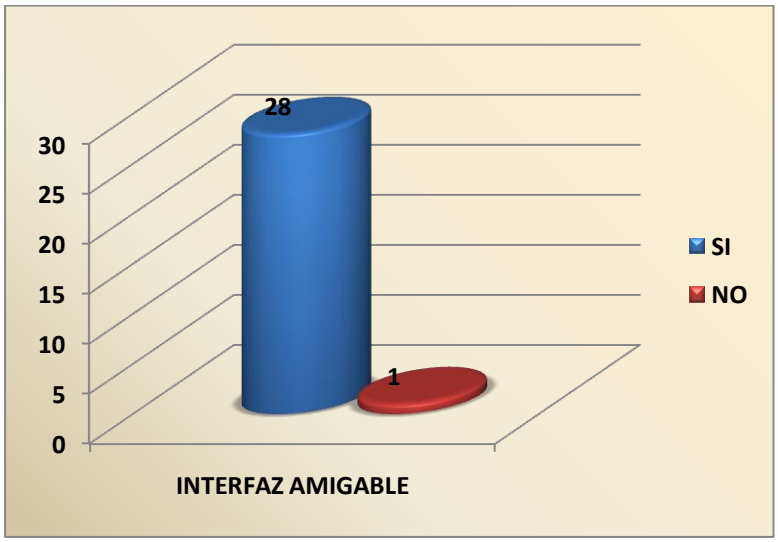


Figura 14. Pregunta Nro. 1 (Interfaz Amigable)

2. **¿Cree usted que el Diseño de la aplicación es conveniente para facilitar las actividades académicas que usted realiza?**

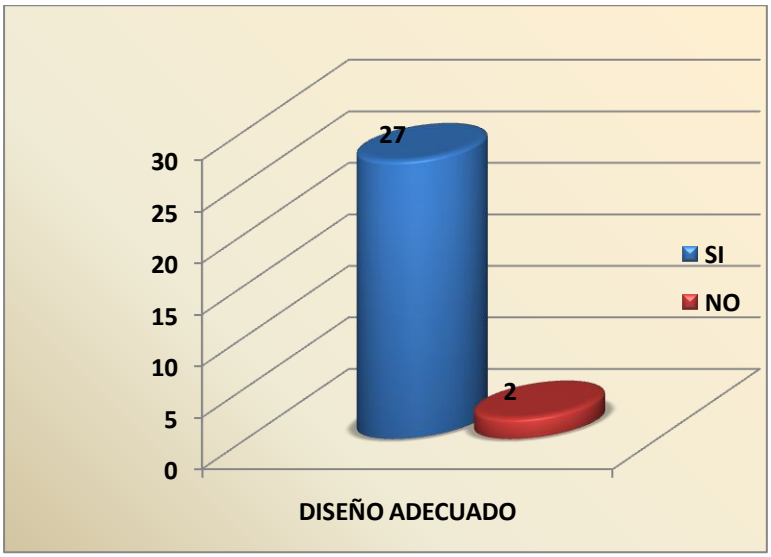


Figura 15. Pregunta Nro. 2 (Diseño Adecuado)

3. **¿Piensa usted que la información presentada en la aplicación es suficiente para el desarrollo de sus actividades académicas?**

Figura 16. Pregunta Nro. 3 (Útil para actividades)

4. **¿Tuvo algún inconveniente al utilizar la aplicación?**

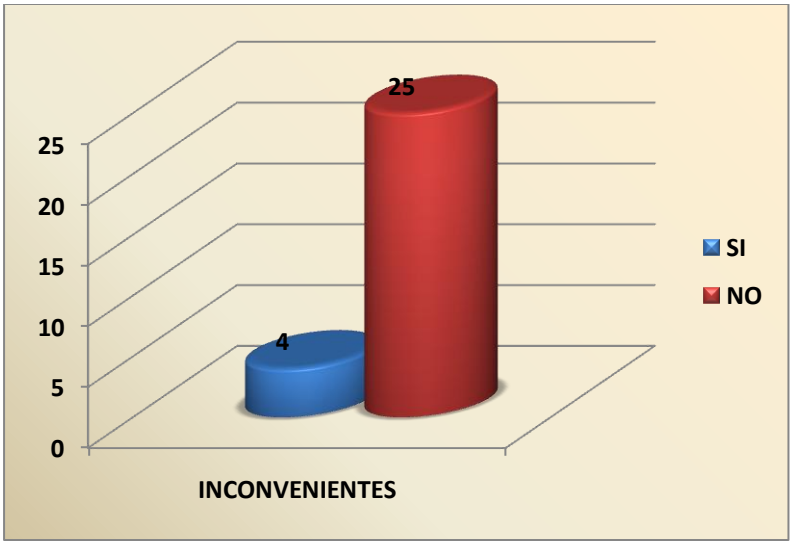


Figura 17. Pregunta Nro. 4 (Inconvenientes)

5. **¿Considera razonable el tiempo de ejecución de la aplicación?**

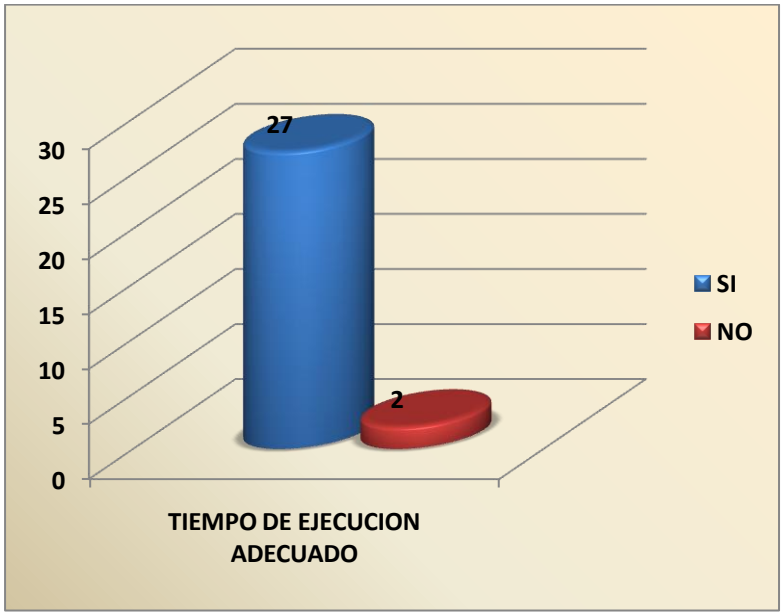


Figura 18. Pregunta Nro. 5 (Tiempo de ejecución adecuado)

6. **¿Considera que la aplicación es útil en cuanto a la agilización del proceso de corrida de algoritmos y detección de errores?**

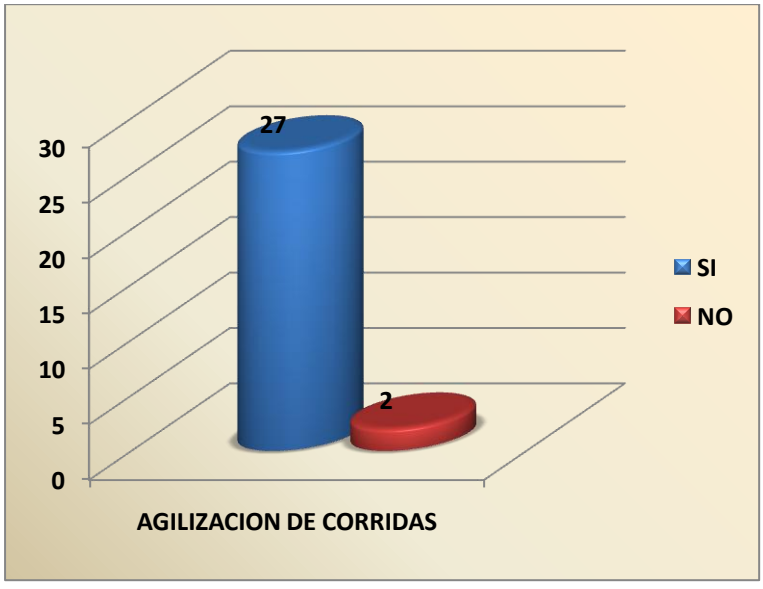


Figura 19. Pregunta Nro. 6 (Agilización de corridas)

7. **¿Cree usted que el almacenamiento de información que ofrece la aplicación es rápido y seguro?**

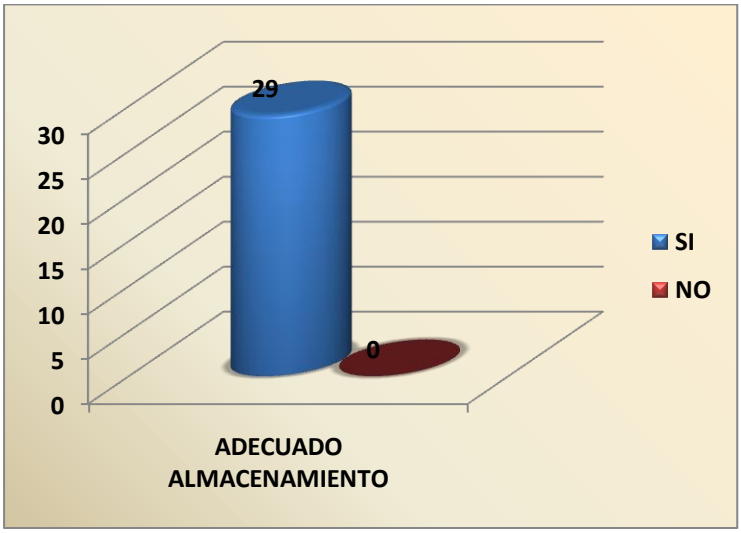


Figura 20. Pregunta Nro. 7 (Adecuado Almacenamiento)

8. **¿Piensa usted que con la aplicación podrá practicar más y mejorar los conocimientos adquiridos en el aula?**

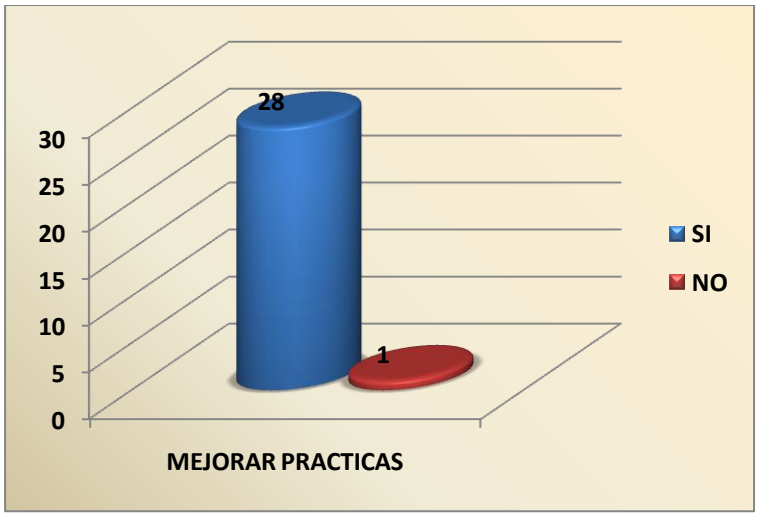


Figura 21. Pregunta Nro. 8 (Mejorar Prácticas)

F.1.5.3. Informe de Resultados de Pruebas de Validación

Para realizar la validación, la aplicación fue probada por la Ing. Mireya Erreyes, docente de la unidad de Metodología de la Programación y alumnos de Quinto Módulo de la carrera de Ingeniería en Sistemas de la Universidad Nacional de Loja.

Informe de: Navegabilidad y funcionamiento de ARK
Fecha: 26 de Octubre de 2011

Tabla 11. Informe de Resultados de Pruebas de Validación

IDENTIFICADOR	Funcionamiento de ARK
RESUMEN	Las pruebas se realizaron con la participación de la Ing. Mireya Erreyes, actual encargada de las unidades de Metodología de la programación y Estructura de Datos, y con 28 alumnos del Quinto Modulo paralelo "C" el día miércoles 26 de octubre de 2011 de 08H00 a 09H00 de la mañana.
VARIACIONES	Se realizó una pequeña capacitación sobre el funcionamiento de ARK.
RESUMEN DE RESULTADOS	Luego de realizar la capacitación se procedió a proporcionar el instalador a los estudiantes y docente para que realizaran pruebas en sus máquinas, además se realizó un ejemplo práctico con un estudiante en el proyector para que todos lo observaran y pudieran comprobar el funcionamiento de la aplicación.
APROBACIÓN	La Ing. Mireya Erreyes aprobó la aplicación luego de hacer las pruebas correspondientes, de igual manera se obtuvo una total aceptación por parte de los estudiantes.

Según los resultados obtenidos al aplicar las pruebas de validación no se tuvo mayores sugerencias ni críticas sobre los fallos o incumplimiento de requerimientos, por lo que la fase de pruebas queda concluida y se da por aceptada la aplicación desarrollada (ANEXO C).

G. DISCUSIÓN

G.1. Metodología XP (Extreme Programing)

G.1.1. Historia de Usuario

Las historias de usuario son la técnica utilizada en XP para especificar los requisitos del software[9].

Tabla 12. Historia de Usuario

<table>
<tr><td colspan="3">Historia de Usuario</td></tr>
<tr><td>Número: 001</td><td colspan="2">Nombre Historia de Usuario: Funcionamiento de Compilador</td></tr>
<tr><td colspan="2">Usuario: Estudiante de Quinto Modulo de Ingeniería en Sistemas de la Universidad Nacional de Loja</td><td rowspan="3">Programadores Encargados:
• Alex Román
• Alondra Ordóñez</td></tr>
<tr><td colspan="2">Prioridad en Negocio: Alta</td></tr>
<tr><td colspan="2">Riesgo en Desarrollo: Bajo</td></tr>
<tr><td colspan="3">Descripción: Es necesaria una herramienta de apoyo para el aprendizaje de programación, que facilite el desarrollo de algoritmos, permitiendo verificar los resultados y ayudando a detectar más fácil los problemas, esta herramienta deberá:
• Facilitar el manejo de la información, podría ser mediante la administración de archivos.</td></tr>
</table>

[9]UNIVERSIDAD LOS ANGELES DE CHIMBOTE, Sistemas de información II, [ftp://200.60.110.5/Docentes/ Orlando_Iparraguirre/2008/SI-II/Sesion%2014/historias.pdf]

• Ayudar para realizar algoritmos de manera más rápida y así aprovechar mejor el tiempo en clases. • Aceptar sentencias en español para facilitar el trabajo, ya que sería mucho más sencillo practicar en español que en inglés. • Brindar la ayuda suficiente para poder utilizarla sin problema. • Proporcionar una interfaz amigable, fácil de manejar. • Permitir ser utilizada en diferentes sistemas operativos.
Observaciones: Cabe mencionar que tanto el docente como los alumnos serán usuarios de la aplicación por lo que los requerimientos son iguales en ambos casos.

G.1.2. Tarjetas CRC (Clase – Responsabilidad - Colaborador)

Tabla 13. Tabla CRC de Clase Compilador

Nombre de la clase: Compilador	
Responsabilidades: • Realizar un Analizador Léxico. • Realizar un Analizador Sintáctico. • Realizar un Analizador Semántico. • Realizar la Generación de Código.	**Colaboradores:** • Alex Román • Alondra Ordóñez • ANTLR

Tabla 14. Tabla CRC de Usuario

Nombre de la clase: Usuario	
Responsabilidades: • Interactuar con el software. • Ingresar algoritmos de pseudocódigo en español y los ejecuta. • Administra archivos.	**Colaboradores:** • Manual de Usuario

Tabla 15. Tabla CRC de Ventana Principal

Nombre de la clase: Ventana Principal	
Responsabilidades: • Permite la interacción con el Usuario. • Permite ingresar algoritmos de pseudocódigo en español y ejecutarlos, mostrando resultados en consola. • Permite la administración de Archivos. • Ofrece ayuda sobre su funcionamiento.	**Colaboradores:** • Usuario • Java JDK v1.6

G.1.3. Requerimientos

Determinar requisitos consiste en estudiar un sistema para conocer cómo trabaja y donde es necesario efectuar mejoras[10].

G.1.3.1. Requerimientos Funcionales

El sistema permitirá:

Tabla 16. Requerimientos Funcionales

CODIGO	DETALLE	CATEGORIA
RF001	Crear, abrir, guardar y editar archivos fuente.	E
RF002	Compilar archivos fuente	O
RF003	Generar código intermedio luego de la compilación	O
RF004	Ejecutar código intermedio	O
RF005	Realizar un análisis léxico que reconoce lexemas válidos y genera tokens.	O
RF006	Realizar un análisis sintáctico en base al analizador léxico.	O
RF007	Realizar un análisis semántico en base al analizador sintáctico.	O
RF008	Ofrecer una ayuda para el aprendizaje de algoritmos.	E

[10] Identificación de Requerimientos [http://www.mitecnologico.com/Main/IdentificacionDeRequerimientos]

G.1.3.2. Requerimientos No Funcionales

Tabla 17. Requerimientos no Funcionales

CODIGO	DETALLE
RNF001	El sistema será desarrollado bajo plataforma JAVA
RNF002	El sistema será fácil de manejar utilizando interfaces graficas ricas para escritorio.
RNF003	El sistema será multiplataforma
RNF004	El sistema tendrá temas de ayuda que permitan una mejor navegabilidad.

G.1.3.3. Glosario de Términos

Usuario (actor): La persona que va a hacer uso del sistema.
Archivos fuente: Lugar donde se almacena el código fuente.
Código intermedio: Es el resultado de compilar al archivo fuente.
Análisis Léxico: Reconoce lexemas válidos y genera tokens.
Lexemas: Secuencia de símbolos que deben estar incluidos en el lenguaje.
Tokens: Identificador de un lexema.
Análisis Sintáctico: Verifica que se cumplan las reglas establecidas.
Análisis Semántico: Verifica que las sentencias tengan sentido.
Código Fuente: Contiene sentencias en pseudocódigo.
Ayuda: Ofrece ayuda al usuario.

G.1.3.4. Modelo de Dominio

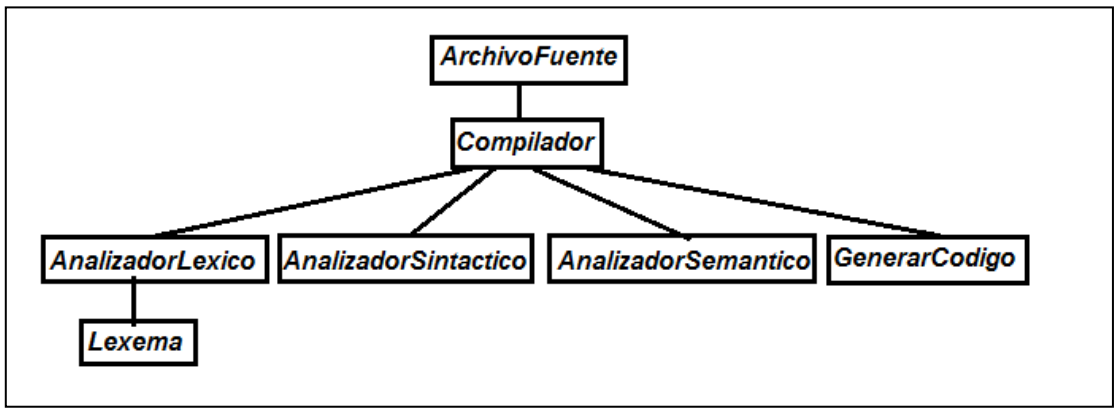


Figura 22. Modelo de Dominio

G.1.4. Diagrama de Clases Final

Diagrama de Paquetes

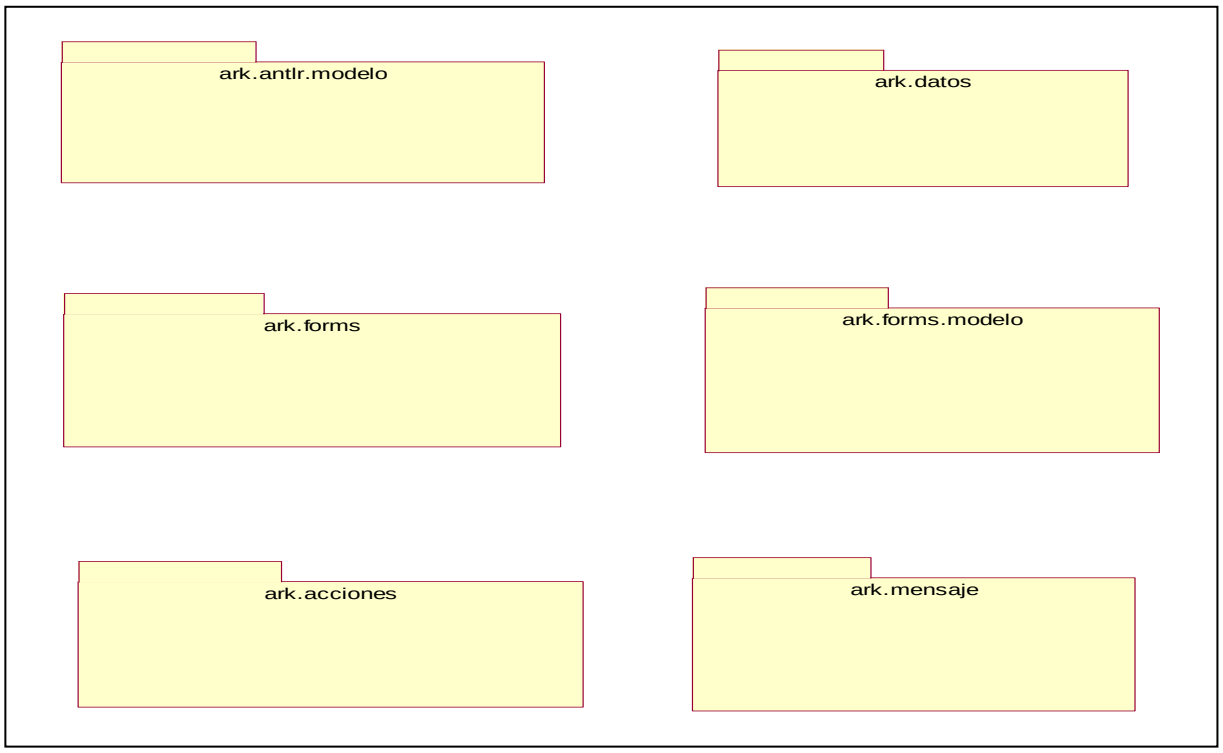


Figura 23. Diagrama de Paquetes

Paquete ARK.ANTLR.MODELO

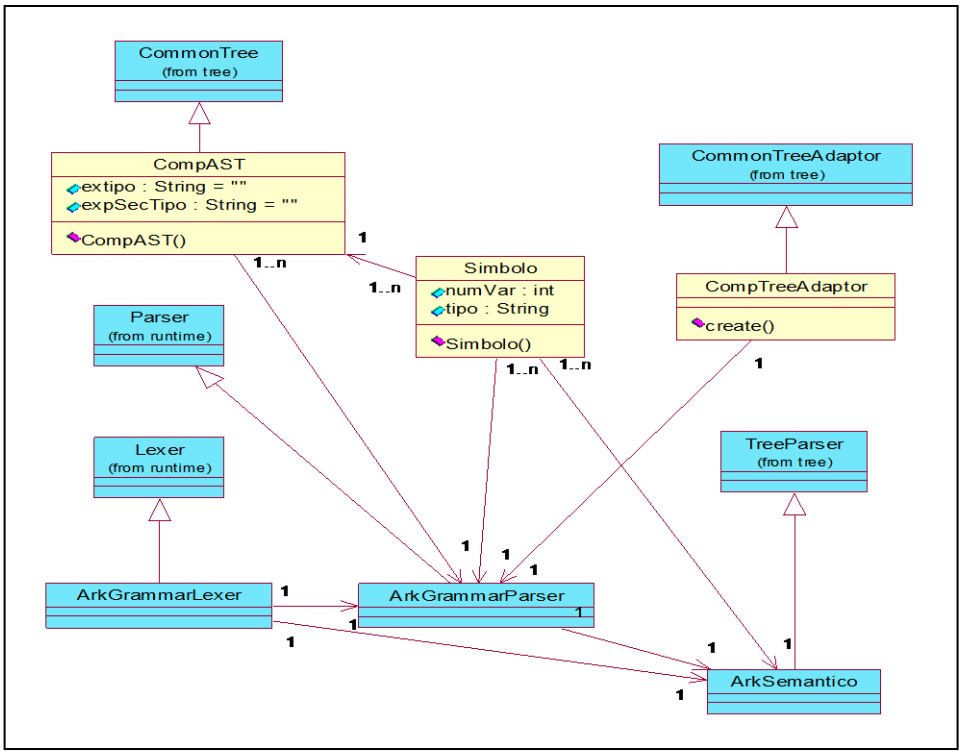


Figura 24. Paquete ARK.ANTLR.MODELO

Paquete ARK.DATOS

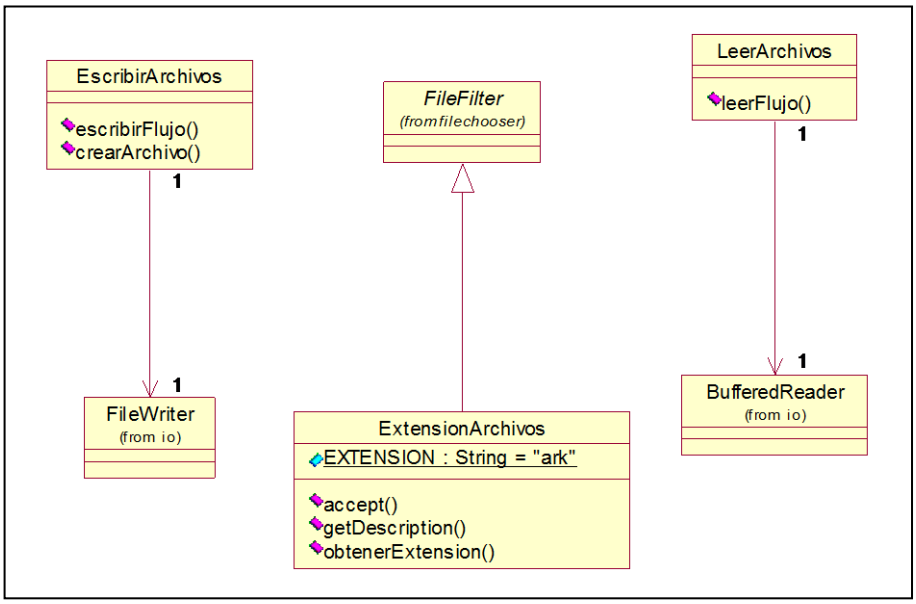


Figura 25. Paquete ARK.DATOS

Paquete ARK.FORMS

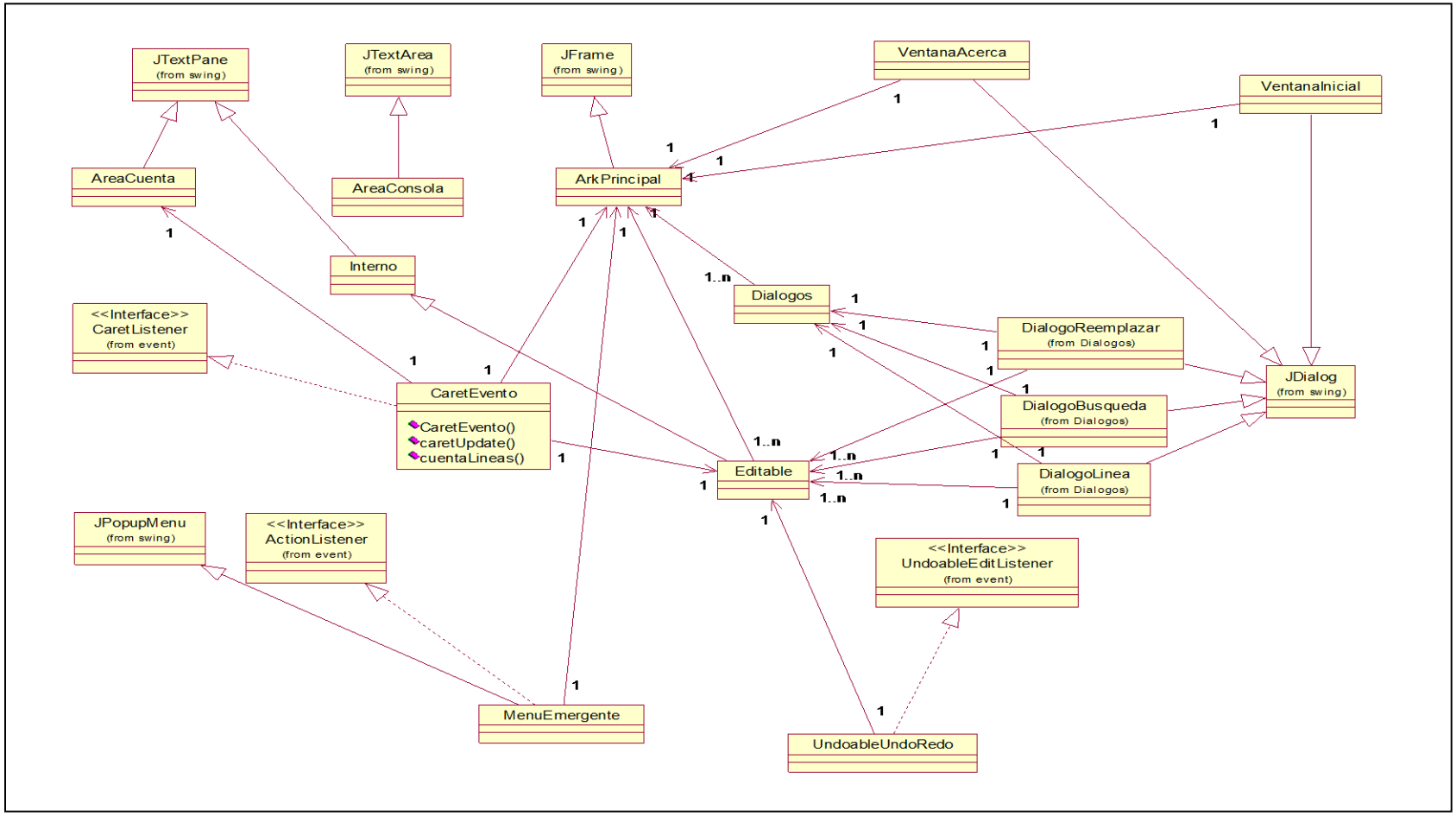


Figura 26. Paquete ARK.FORMS

Paquete ARK.FORMS.MODELO

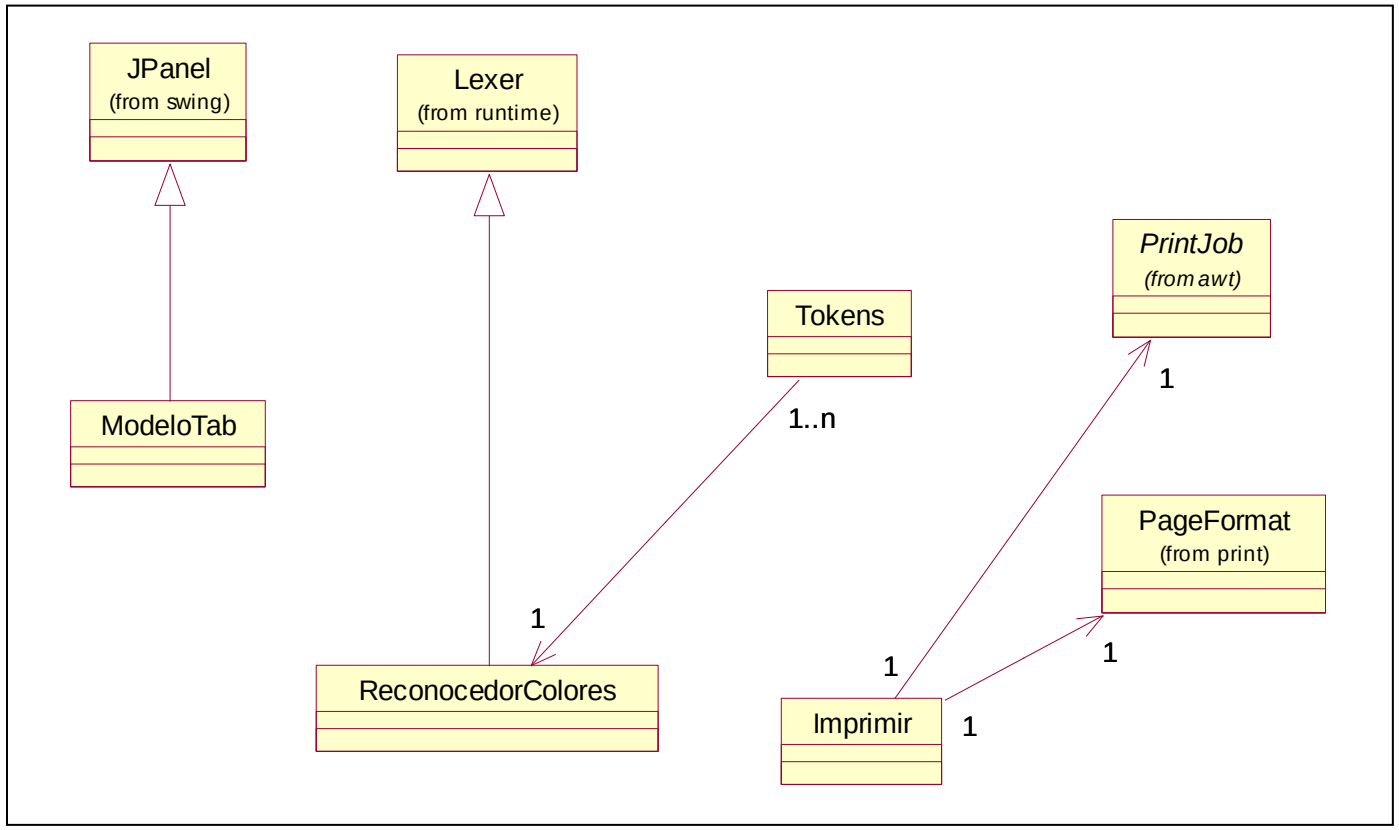


Figura 27. Paquete ARK.FORMS.MODELO

Paquete ARK.ACCIONES

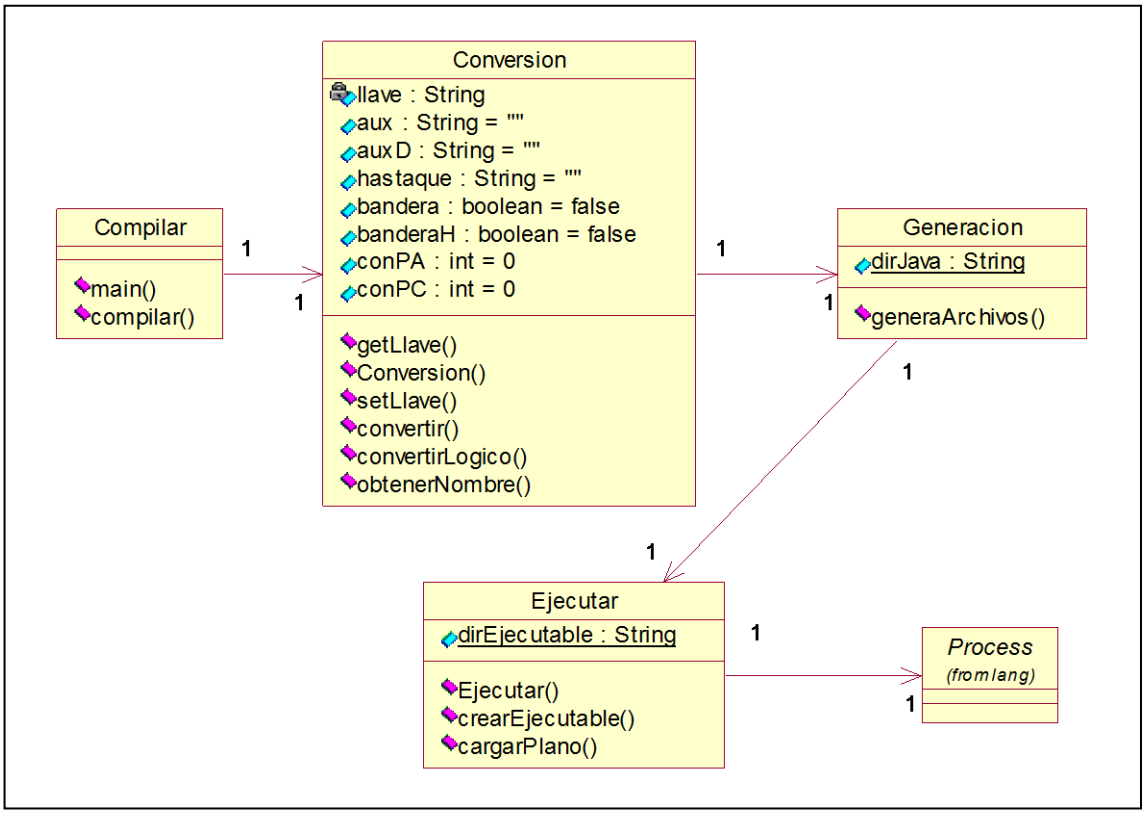


Figura 28. Paquete ARK.ACCIONES

Paquete ARK.MENSAJE

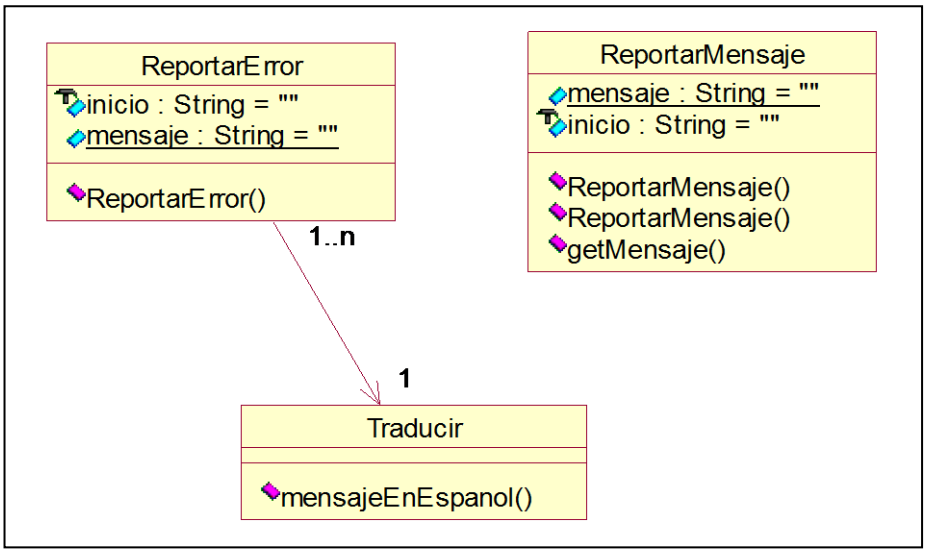


Figura 29. Paquete ARK.MENSAJE

G.2. Valoración Técnico Económica Ambiental

Una vez terminado el desarrollo de la aplicación y luego de haber aplicado las pruebas se considera que el compilador ARK es un sistema con características esenciales para resolver y encaminar de mejor manera las actividades que se realiza en la Unidad de Metodología de la Programación.

Tabla 18. Valoración Técnico Económica Ambiental

DESCRIPCIÓN	CANTIDAD	N° DE HORAS	COSTOS UNITARIOS	COSTO TOTAL
1. RECURSOS HUMANOS				
Aspirantes a Ingeniería en Sistemas	2	800	$4	$ 3200
Director de Tesis	1	0	0	0
2. RECURSOS TECNICOS				
2.1 Hardware				
Computadores	2	400	0.30	$ 120
Impresora HP D1560	1	10	0.50	$ 5
2.2 Software				
NetBeans IDE v7.0	2		Gratuito	$ 0
Eclipse Helios	2		Gratuito	$ 0
Java JDK v1.6	2		Gratuito	$ 0
JHelp	2		Gratuito	$ 0
ANTLR v3.2	2		Gratuito	$ 0
ANTLRWorks v1.3.1	2		Gratuito	$ 0
2.3 Recursos Académicos				
TEXTO: The Definitive the ANTLR Reference	1		$ 24	$ 24
2.3 Recursos Materiales				
Resmas de Papel	4		$4	$ 16

Bolígrafos	3		$0.2	$ 0.6
Cartuchos de tinta	2		$20	$ 40
Internet/horas	50		$0.7	$ 35
2.4 Gastos Generales				
Luz	800		$ 0.50	$ 400
Teléfono	300		$ 1	$ 300
Transporte	1000		$ 0.25	$ 250
			SUBTOTAL	$ 4390,60
			IMPREVISTOS 5%	$ 219,53
			COSTO TOTAL	$ 4610,13

H. CONCLUSIONES

- ✓ En el análisis léxico se pudo determinar las palabras reservadas, los separadores, operadores y todos los lexemas que conforman el lenguaje ARK.
- ✓ En el análisis sintáctico se estableció las reglas gramaticales del compilador, que permiten controlar la estructura de las instrucciones escritas en lenguaje ARK.
- ✓ Se construyó un analizador semántico en el que se establece reglas semánticas lo que permite que el lenguaje tenga sentido.
- ✓ La generación de Código intermedio permite la ejecución del código fuente, previa la comprobación de errores.
- ✓ La aplicación fue evaluada por los usuarios, luego de la respectiva capacitación para comprobar su validez, los cuales afirmaron que la aplicación cumple con todos los requerimientos y la aprobaron en su totalidad.
- ✓ ANTLR presenta eficiencia en la recuperación de errores gracias al recorrido descendente recursivo que realiza.
- ✓ ANTLR permite la construcción de AST (Abstract Syntax Tree) durante el establecimiento de las reglas gramaticales en el lenguaje para posteriormente ser utilizados en el análisis semántico a través del recorrido descendente recursivo, por lo que es de gran ayuda al momento de establecer las reglas semánticas del lenguaje.
- ✓ El software permite crear, abrir y guardar archivos permitiendo un mejor aprovechamiento de la información por parte del usuario.
- ✓ Mediante la elaboración de ARK tanto el docente como los estudiantes de Cuarto y Quinto Modulo de Ingeniería en Sistemas de la Universidad Nacional de Loja agilizarán y mejorarán el proceso de enseñanza-aprendizaje de la unidad de Metodología de la Programación.

I. RECOMENDACIONES

- ✓ Para la creación de un compilador se recomienda determinar todos los lexemas del lenguaje, ya que sirven como base para las demás fases del compilador.
- ✓ Es recomendable establecer las reglas gramaticales de acuerdo a lo que necesita el lenguaje, teniendo en cuenta que la estructura sea la adecuada.
- ✓ En la construcción del analizador semántico se recomienda considerar todos los posibles errores para poder establecer el debido control.
- ✓ La generación de código se realiza una vez que estén completas las fases anteriores, por lo que se recomienda implementarlas cuidadosamente, verificando que estén correctas antes de proceder con la generación de código, para evitar posible pérdida de tiempo.
- ✓ Una vez desarrollado el compilador se recomienda validar con los usuarios, para que sean ellos quienes determinen si está acorde a sus necesidades, caso contrario realizar las respectivas correcciones.
- ✓ Es importante conocer y hacer uso de las herramientas que hoy en día existen para facilitar y mejorar el desarrollo de compiladores como es el framework ANTLR v 3.2.
- ✓ Se recomienda la utilización de ARK en las actividades académicas de los estudiantes de Ingeniería en Sistemas, sobre todo de los módulos iniciales, tanto dentro como fuera del aula, ya que es de gran ayuda en cuanto a la ejecución de algoritmos, administración de información y facilita el proceso de aprendizaj**e.**

J. BIBLIOGRAFÍA

2.8.1. Libros

- ✓ Aho, Alfred V.; Ullman, Jeffrey D.; COMPILADORES: PRINCIPIOS, TÉCNICAS Y HERRAMIENTAS.
- ✓ Alfonseca Moreno, Manuel; Echeandía, Marina de la Cruz, Ortega de la Puente, Alfonso; Pulido Cañabate, Estrella. COMPILADORES E INTERPRETES: teoría y práctica, Pearson Prentice Hall, 2006.
- ✓ Castro Esteban, David. TEORÍA DE AUTÓMATAS, LENGUAJES FORMALES Y GRAMÁTICAS. Universidad de Alcalá. 2003-2004.
- ✓ Louden, Kenneth C. CONSTRUCCIÓN DE COMPILADORES PRINCIPIOS Y PRÁCTICA. International Thomson Editores. UNED. 2004.
- ✓ Louden, K.C, CONSTRUCCIÓN DE COMPILADORES*: Principlos y Práctica*, 1997, Tema 2
- ✓ Louden, K.C, CONSTRUCCIÓN DE COMPILADORES: PRINCIPLOS Y PRÁCTICA, 1997, Tema 2, páginas: 31-93.
- ✓ Luengo, Cándida, ANÁLISIS SINTÁCTICO EN PROCESADORES DE LENGUAJE, Oviedo, Enero 2005.
- ✓ Llano Diaz, Emiliano. ANÁLISIS Y DISEÑO DE COMPILADORES. México. Primera Edición. 2002.
- ✓ Martínez Gloria, García Luis A. APUNTES DE TEORÍA DE AUTÓMATAS Y LENGUAJES FORMALES. 2005.
- ✓ PARR, Terence. A quick tour for the impatient, THE DEFINITIVE ANTLR REFERENCES, BUILDING DOMAIN-SPECIFIV LANGUAGE, Dallas, Texas, pp. 59-84. 2010.
- ✓ Rojas Gálvez, Sergio; Mora Mata, Miguel Ángel. COMPILADORES: TRADUCTORES Y COMPILADORES CON LEX/YACC, JFLEX/CUP Y JAVACC. Universidad de Málaga. 2005.

2.8.2. Textos Electrónicos

- ✓ Universidad los Ángeles de Chimbote, SISTEMAS DE INFORMACIÓN II, [http://200.60.110.5/Docentes/Orlando_Iparraguirre/ 2008/SI-II/Sesion% 2014/historias.pdf]|
- ✓ Morales Luna Guillermo, Árboles de Derivación, 2000, [http://delta.cs.cinvestav.mx/~gmorales/ta/node104.html]
- ✓ Autómatas Finitos [http://es.wikipedia.org/w/index.php?title=Aut%C3%B3mata_finito&amp]
- ✓ Lenguaje regular [http://es.wikipedia.org/w/index.php?title=Lenguaje_regular&amp]
- ✓ Gramática Libre de Contexto [http://es.wikipedia.org/w/index.php?title=Gram%C3%A1tica_libre_de_contexto&amp]
- ✓ ANother Tool for Language Recognition, 2010. [http://es.wikipedia.org/wiki/ANTLR]
- ✓ Brena, Ramón. AUTÓMATAS Y LENGUAJES. Tecnológico de Monterrey. 2003. [http://homepages.mty.itesm.mx/rbrena/AyL.html].
- ✓ Identificación de Requerimientos [http://www.mitecnologico.com /Main/IdentificacionDeRequerimientos]
- ✓ Depto. de Informática Universidad de Valladoil, Teoría de autómatas y lenguajes formales, 2009 URL: [http://www.infor.uva.es/~teodoro/Ejer3WEB.pdf].
- ✓ Árbol de sintáxis Abstracta, [http://www.uco.es/~ma1fegan/Comunes /manuales/pl/ANTLR/Arboles-de-sintaxis-abstracta.pdf]
- ✓ Sánchez Isabel, Cárdenas María Antonia, Teoría de autómatas y lenguajes formales, URL: [http://www.yakiboo.net/apuntes/TALF%20-%20YakiBoo.net.pdf].

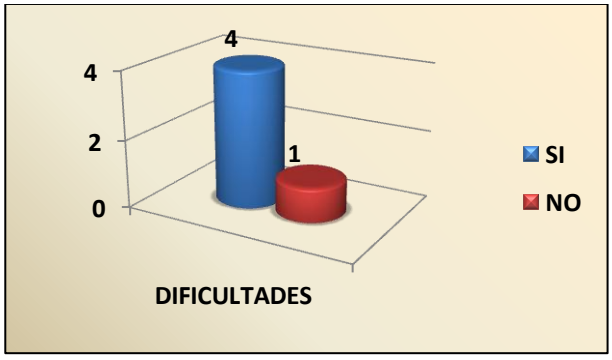
K. ANEXOS

2.8.3. Anexo A

TABULACION DE ENTREVISTA REALIZADA A LOS ALUMNOS DE QUINTO MÓDULO PARA OBETENER LOS REQUERIMIENTOS DEL SISTEMA

1. **¿Tuvo alguna dificultad en el transcurso de la unidad de Metodología de la Programación?**

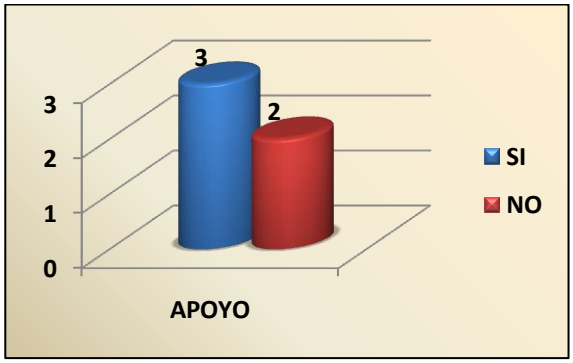
Respuestas	Valor	Porcentaje
SI	4	80%
NO	1	20%
TOTAL	5	100%



La mayoría de los entrevistados afirmaron que sí tuvieron problemas en la unidad de Metodología de la Programación debido a una mala metodología, a falta de explicación por parte del docente y porque no se cumplió con la carta descriptiva.

2. **¿Utilizó algún tipo de software de apoyo a la materia?**

Respuestas	Valor	Porcentaje
SI	3	60%
NO	2	40%
TOTAL	5	100%



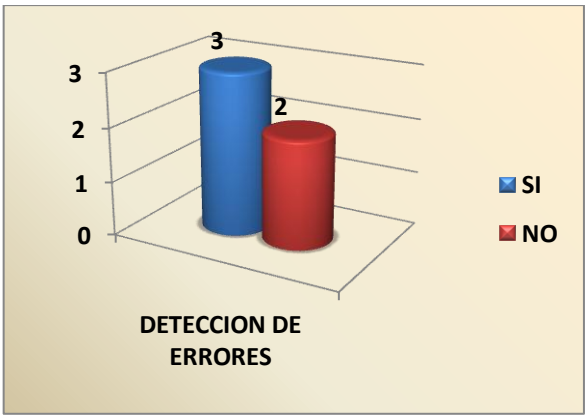
Los programas que utilizaban como apoyo de la materia fueron Pseint, netbeans y Dia-Edt.

3. **¿Emplea mucho tiempo en la ejecución (corrida) a mano de algoritmos?**

Respuestas	Valor	Porcentaje
SI	0	0%
NO	5	100%
TOTAL	5	100%

4. **¿Podía encontrar los errores cometidos en un algoritmo de manera rápida?**

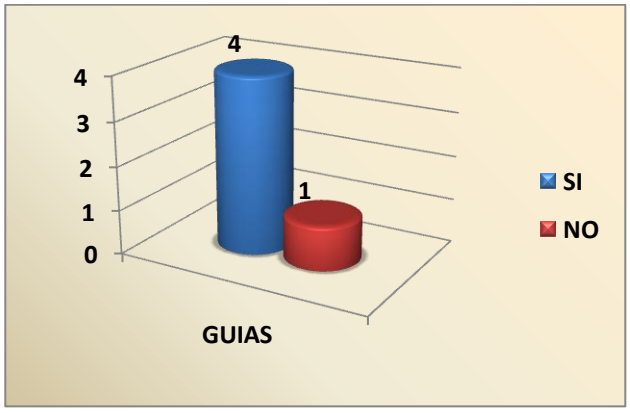
Respuestas	Valor	Porcentaje
SI	3	60%
NO	2	40%
TOTAL	5	100%



Hay un pequeño porcentaje de alumnos que no logran detectar los errores de manera rápida por lo que necesitan ayuda en este aspecto.

5. **¿Contó con una(s) guía(s) a parte del docente para el desarrollo de algoritmos?**

Respuestas	Valor	Porcentaje
SI	4	80%
NO	1	20%
TOTAL	5	100%



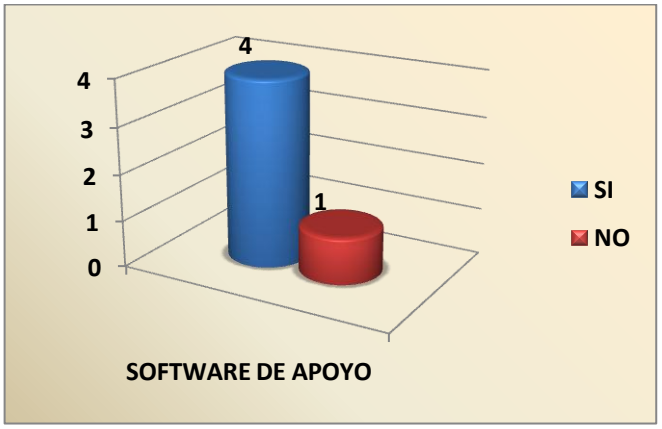
Los entrevistados afirman que si contaban con ayudas extra como folletos propios e información de Internet.

6. **¿Tenía problemas con la forma de almacenar los trabajos realizados?**

Respuestas	Valor	Porcentaje
SI	0	0%
NO	5	100%
TOTAL	5	100%

7. **¿Cree Ud. que es necesario utilizar un software que sirva de apoyo para facitar el aprendizaje de esta materia?**

Respuestas	Valor	Porcentaje
SI	4	80%
NO	1	20%
TOTAL	5	100%



La mayoría de entrevistados creen que si es necesario utilizar un software para apoyo de la unidad por las siguientes razones: Facilita el desarrollo de algoritmos, Permite verificar los resultados y ayudan a detectar más fácil los problemas.

2.8.4. Anexo B

UNIVERSIDAD NACIONAL DE LOJA

AREA DE LA ENERGIA, LAS INDUSTRIAS Y LOS RECURSOS NATURALES NO RENOVABLES

INGENIERIA EN SISTEMAS

Desarrollo de un Compilador en español para la ejecución de algoritmos en pseudocódigo

Integrantes:

Alondra Ordóñez

Alex Román

Coordinadora:

Ing. Ketty Palacios

Modulo y paralelo:

10º Modulo "B"

Loja-Ecuador

2010

1. TEMA

Desarrollo de un Compilador en español para la ejecución de algoritmos en pseudocódigo

2. SITUACIÓN PROBLEMÁTICA

2.1. Antecedentes

La Universidad Nacional de Loja, es una Institución de Educación Superior, laica, autónoma, de derecho público, con personería jurídica y sin fines de lucro, de alta calidad académica y humanística, que ofrece formación en los niveles: técnico y tecnológico superior; profesional o de tercer nivel; y, de postgrado o cuarto nivel; que realiza investigación científico-técnica sobre los problemas del entorno, con calidad, pertinencia y equidad, a fin de coadyuvar al desarrollo sustentable de la región y del país, interactuando con la comunidad, generando propuestas alternativas a los problemas nacionales, con responsabilidad social; reconociendo y promoviendo la diversidad cultural y étnica y la sabiduría popular, apoyándose en el avance científico y tecnológico, en procura de mejorar la calidad de vida del pueblo ecuatoriano.

El Área de Energía, Industrias y Recursos Naturales No Renovables de la Universidad Nacional de Loja, tiene sus inicios en el año 1984, como resultado de la fusión bajo una misma estructura administrativa y académica de algunas carreras técnicas que existían en la anterior Facultad de Ciencias de la Educación y en los Institutos Tecnológicos Universitarios que venían funcionando en algunos cantones de la provincia de Loja.

El Área de la Energía, las Industrias y los Recursos Naturales No Renovables, como una unidad académica joven, dinámica y con visión futurista, viene promoviendo carreras a nivel Terminal y Tecnológico tales como Ingeniería en Sistemas, Ingeniería en Electromecánica, Ingeniería Geología, Ingeniería en Electrónica, Tecnología en Electricidad, Tecnología en Electrónica y Maestría en Electromecánica.[11]

[11]Pagina Web de la Universidad Nacional de Loja, [www.unl.edu.com]

2.2. Situación Problemática

En el Área de Energía, las Industrias y los Recursos Naturales no Renovables de la Universidad Nacional de Loja específicamente en la carrera de Ingeniería en Sistemas el aprendizaje de la Programación es muy importante pues abarca gran parte del campo ocupacional de un Ingeniero en sistemas, es por esta razón que desde el cuarto modulo se imparten unidades que ayudan a los estudiantes a introducirse en el maravilloso mundo de la programación, tales como Metodología de la Programación, Programación Básica, Programación avanzada, Estructura de Datos, Lenguaje Ensamblador, Ingeniería del Software, Simulación e Inteligencia Artificial, además de otras unidades que sirven como complemento a los conocimientos adquiridos por el estudiante.

La programación es un proceso lógico que ayuda a resolver problemas de toda índole a través de programas que contienen algoritmos que proporcionen una solución para estos problemas, por esta razón para iniciar en el aprendizaje de la programación es importante aprender en primer lugar a diseñar algoritmos utilizando un falso lenguaje llamado pseudocódigo. El pseudocódigo es una descripción de alto nivel de un algoritmo que emplea una mezcla de lenguaje natural con algunas convenciones sintácticas propias de lenguajes de programación, como asignaciones, ciclos y condicionales, entre otras. El pseudocódigo está pensado para facilitar a las personas el entendimiento de un algoritmo, y por lo tanto puede omitir detalles irrelevantes que son necesarios en una implementación. El pseudocódigo en general es comprensible y a la vez suficientemente estructurado para que su implementación se pueda hacer directamente a partir de él.[12]

La importancia de aprender a elaborar algoritmos en pseudocódigo radica en que estos algoritmos sirven para resolver desde los problemas más sencillos hasta un complejo problema de inteligencia artificial, lo que implica que no sólo estarán presentes a lo largo de toda la carrera universitaria, sino también serán de mucha ayuda en el desempeño de las labores profesionales; por esta razón es importante

[12] Pseudocódigo [http://es.wikipedia.org/wiki/Pseudoc%C3%B3digo]

que a los estudiantes no se les dificulte el aprendizaje y que no acumulen vacíos en el conocimiento, sobre todo en los primeros módulos en los que afianzan sus bases para módulos posteriores.

En base a lo mencionado anteriormente podemos darnos cuenta que hay algunos problemas en las primeras unidades de enseñanza de la carrera de Ingeniería en Sistemas, los cuales mencionamos a continuación:

- Pérdida de tiempo al ejecutar algoritmos en pseudocódigo manualmente, por lo que las dos horas diarias que están dispuestas para cada unidad no son suficientes y no avanzan a abarcar todos los contenidos planificados inicialmente.
- Dificultad para detectar errores en la elaboración de algoritmos, por lo que les lleva más tiempo resolver un problema.
- Falta de una ayuda para la construcción de algoritmos en pseudocódigo, que le sirva de guía al estudiante cuando no puedan recibir asesoría por parte del docente.
- Falta de práctica en el aula y en casa por parte de los estudiantes por no tener una herramienta que les ayude a realizar sus prácticas de manera más fácil.
- Dificultad para almacenar de manera ordenada los algoritmos elaborados o las prácticas elaboradas, lo que retrasa al estudiante cuando intenta buscar trabajos realizados anteriormente para reutilizar código.

El presente proyecto está enfocado a resolver estos problemas que podrían presentarse en cualquier institución educativa, por lo que el presente trabajo que pretende proporcionar una solución para la Universidad Nacional de Loja, será seguramente una solución a los problemas de otras entidades educativas.

2.3. Problema de investigación

Por todo lo mencionado anteriormente y con el afán de mejorar la calidad de enseñanza creemos que el problema a solucionar es: **"Falta de una herramienta de software para la ejecución de programas en pseudocódigo para los**

alumnos de cuarto y quinto modulo de la carrera de Ingeniería en Sistemas de la Universidad Nacional de Loja", por lo que consideramos que es necesaria la implementación de un Compilador en español para la ejecución de algoritmos en pseudocódigo que ayude a solucionar este problema, permitiendo que los alumnos practiquen más y se den cuenta de sus errores para corregirlos, de esta manera se agiliza y mejora el proceso de enseñanza-aprendizaje.

2.4. DELIMITACIÓN

Implementar un Compilador en español para la ejecución de algoritmos en pseudocódigo utilizando el framework ANTLR.

2.4.1. Problemas específicos de investigación

- Pérdida de tiempo al ejecutar algoritmos en pseudocódigo manualmente.
- Dificultad para detectar errores en la elaboración de algoritmos.
- Falta de una ayuda para la construcción de algoritmos en pseudocódigo.
- Falta de práctica en el aula y en casa por parte de los estudiantes.
- Dificultad para almacenar de manera ordenada los algoritmos elaborados o las prácticas elaboradas.

2.4.2. Espacio

Todos los problemas mencionados anteriormente tienen lugar en el Cuarto y Quinto Módulo de la carrera de Ingeniería en Sistemas de la Universidad Nacional de Loja, por lo que el presente proyecto se realizará en este lugar.

2.4.3. Tiempo

El tiempo de desarrollo de la presente tentativa es de un año.

2.4.4. Unidades de observación

- Estudiantes de Cuarto y Quinto Módulo de la carrera de Ingeniería en Sistemas de la Universidad Nacional de Loja.

- Docentes de la carrera.
- Unidades de metodología de la Programación y Estructura de Datos de cuarto y quinto módulo respectivamente.

3. JUSTIFICACIÓN

La Universidad Nacional de Loja, es una institución educativa de gran reconocimiento a nivel nacional e internacional, cuya preocupación es preparar y educar a las personas, quienes ingresan sus aulas formándose con conocimientos teóricos y prácticos que le permita dar alternativas de solución a los diferentes problemas que tiene la sociedad.

La Universidad Nacional de Loja está conformada por cinco áreas orientadas a la formación de profesionales, entre las áreas que conforman esta institución, destacaremos el Área de Energía, Industrias y Recursos Naturales no Renovables cuyas actividades encaminan al crecimiento, enriquecimiento y desarrollo de entes sociales que reciben una educación científica, técnica y tecnológica en diferentes carreras siendo de nuestro interés la carrera de Ingeniería en Sistemas en la cual se forma profesionales con conocimientos profundos de la estructura y particularidades de hardware, software y comunicaciones, para llevar a la práctica todos estos conocimientos y dar solución a los problemas de la colectividad en general.

3.1. Justificación Académica

La Universidad Nacional de Loja a través del SAMOT, propicia el logro de aprendizajes significativos en sus estudiantes, la formación académica debe estar basada en la teoría y fomentarse en la práctica de tal manera que los alumnos puedan desenvolverse de forma eficiente en el futuro, de la mima manera los estudiantes de la carrera de Ingeniería en Sistemas debemos forjar un perfil de profesional que nos permita hacer frente a cualquier problema que se presente durante nuestra posterior vida laboral.

En base a lo antes expuesto nos orientamos a vincular la teoría con la práctica, aplicando todos los conocimientos adquiridos en los años de estudio universitarios mediante la Implementación de un Compilador en español para la ejecución de algoritmos en pseudocódigo.

3.2. Justificación Económica

Un paso importante antes de iniciar un proyecto es realizar un estudio de factibilidad económica, ya que es una de las variables que afecta a la resolución de los proyectos.

Todos los gastos que implica el desarrollo del presente proyecto serán asumidos por quienes conformamos el grupo de trabajo, por lo tanto creemos que el presente proyecto es factible de realizar ya que contamos con los suficientes recursos económicos, humanos, bibliográficos, técnicos y tecnológicos que permitirán finalizar con éxito el proyecto planteado.

3.3. Justificación Social

Dentro del pensum de estudio de la Carrera Ingeniería en Sistemas de la Universidad Nacional de Loja constan varias unidades relacionadas con la Programación y elaboración de algoritmos en pseudocódigo para la resolución de problemas. Creemos que como estudiantes de Ingeniería en Sistemas el utilizar una herramienta que les permita ingresar sus sentencias en pseudocódigo y ejecutarlas sería muy beneficioso, ya que les ahorraría mucho tiempo tanto en la ejecución del algoritmo como en la detección de errores, que son actividades que en la actualidad se realizan manualmente y que llevan mucho tiempo. Por este motivo el presente proyecto constituye un aporte importante ya que está enfocado a la resolución de los problemas antes mencionados, presentando alternativas que mejoren la situación actual.

3.4. Justificación Operativa

Para el desarrollo del presente proyecto contamos con el apoyo de los docentes encargados de impartir las unidades relacionadas con nuestra investigación, así como también contamos con la colaboración de los alumnos de la carrera, quienes están prestos a facilitarnos toda la información necesaria para llevar a cabo nuestro proyecto y culminarlo con éxito.

4. OBJETIVOS

4.1. OBJETIVO GENERAL

- Implementar un Compilador en español para la ejecución de algoritmos en pseudocódigo utilizando el framework ANTLR.

4.2. OBJETIVOS ESPECÍFICOS

- Realizar un analizador léxico, para comprobar que los lexemas y los tokens estén escritos correctamente.
- Realizar un analizador sintáctico, que verifique que las sentencias estén escritas con la sintaxis correcta.
- Realizar un analizador semántico, para determinar que las sentencias estén de acuerdo a las reglas del lenguaje.
- Realizar la generación de código intermedio para la ejecución de algoritmos en pseudocódigo.
- Realizar las pruebas de validación del software con los estudiantes.
- Capacitación a estudiantes y docentes de la Carrera de Ingeniería en Sistemas.

5. ESQUEMA DEL MARCO TEÓRICO

5.1. TEORÍA DE AUTÓMATAS Y LENGUAJES FORMALES

5.1.1. Alfabetos y Lenguajes

5.1.1.1. Alfabetos, palabras y lenguajes

5.1.1.2. Operaciones con palabras

5.1.1.3. Operaciones con lenguajes

5.1.2. Lenguajes Regulares

5.1.2.1. Lenguajes sobre alfabetos

5.1.2.2. Expresiones regulares

5.1.2.3. AFD

5.1.2.4. AFN

5.1.2.5. Transiciones

5.1.2.6. Propiedades de los lenguajes regulares

5.1.3. Lenguajes independientes del contexto

5.1.3.1. Gramáticas regulares

5.1.3.2. Gramáticas independientes del contexto

5.1.3.3. Árboles de derivación

5.1.3.4. Simplificación de GIC

5.1.4. Análisis Léxico

5.2. COMPILADORES

5.2.1. Análisis Sintáctico

5.2.2. Análisis Semántico y Código Intermedio

5.2.3. Generación de Código

5.2.4. Optimización y Generación de Código Final

REVISIÓN LITERARIA

5.1. TEORÍA DE AUTÓMATAS Y LENGUAJES FORMALES.

Lenguajes formales serán meramente símbolos con una gramática formal para agruparlos.
Los autómatas son dispositivos mecánicos, electrónicos o biológicos que en un punto de tiempo están en un estado, o que dado una razón (por ejemplo una señal de entrada) cambia de estado. Por ejemplo reloj mecánico o electrónico, máquina para lavar, todo un ordenador.

En el contexto de esta asignatura autómatas son máquinas matemáticas con estados y funciones de transición (donde se puede añadir entrada, salida, memoria interna modificable, etc.).

5.1.1. ALFABETOS Y LENGUAJES.

5.1.1.1. ALFABETOS, PALABRAS Y LENGUAJES.

ALFABETO

Un alfabeto es un conjunto finito no vacío de símbolos.

$\sum 1 = \{0, 1\}$

$\sum 2 = \{a, b\}$

$\sum 3 = \{na, pa, bra, la\}$

$\sum 4 = \{<HTML>,</HTML>,<BODY>,</BODY>, . . .\}$

$\sum 5 = \{|\}$

$\sum 6 = \{a, ab, aab\}$

- Usamos meta–símbolos (tal como {, }, =, y la coma) para escribir sobre lo que hablamos.
- Desde el contexto siempre será claro, si se trata de un símbolo del alfabeto o si se trata de un meta–símbolo.
- Usamos subíndices para distinguir diferentes alfabetos.

- Usamos normalmente las minúsculas como alfabeto ∑= {a, . . . , z}, en los ejemplos normalmente letras desde el principio del alfabeto.
- Cardinalidad del alfabeto (número de elementos del alfabeto): |∑| > 0, |∑| < ∞

PALABRAS

Una secuencia finita de símbolos de un alfabeto es una **palabra** sobre dicho alfabeto.

∑1 : 0, 1, 00, 01, 11, 000, 1001101

∑2 : a, aa, abb, ababa

∑3 : napa, palabra

∑6 : a, ab, aab, aaab, abab

- Escribimos la palabra vacía, es decir, la palabra que no contiene ningún símbolo, como ε.
- Usamos normalmente letras minúsculas para anotar palabras, preferiblemente desde el final del alfabeto.
- El símbolo ε no pertenece a ningún alfabeto, ε ∉ ∑

LENGUAJE

Un **lenguaje** es cualquier subconjunto del universo sobre algún alfabeto, es decir L⊂ W(∑), o también L ⊂ ∑*.

Ejemplo

- Lenguajes triviales
  - L = ϕ es el lenguaje vacio (que no contiene ninguna palabra), |L|=0
  - L = {ε} es el lenguaje que solamente contiene la palabra vacio, |L|=1

  Son independientes del alfabeto y por eso son lenguajes sobre cualquier alfabeto.

- Sea ∑ = {a, b}
  - L_1 = {ε, a, b}

- $L_{ab} = \{a^n b^n \mid n \in IN\}$ es decir, el lenguaje que contiene todas las palabras con un número de a's seguidos por el mismo número de b's.
- $L_{pal} = \{ww^R \mid w \in \sum^* \}$ es decir, palíndromos.
- $L_{quad} = \{a^{n2} \mid n \in IN>0\}$
- Si $|L| < \infty$ para un lenguaje $L \subset \sum^*$, entonces se llama L lenguaje finito.

5.1.1.2. OPERACIONES CON PALABRAS.

- La **longitud** de una palabra sobre un alfabeto es el número de símbolos que contiene.

  $\sum 1 : w = 0 \Rightarrow |w| = 1, w = 1001101 \Rightarrow |w| = 7$

  $\sum 2 : w = a \Rightarrow |w| = 1, w = ababa \Rightarrow |w| = 5$

  $\sum 3 : w = napa \Rightarrow |w| = 2, w = palabra \Rightarrow |w| = 3$

  $\sum 6 : w = ab \Rightarrow |w| = 2, w = aab \Rightarrow |w| = 1 \text{ o } |w| = 2$??

  - ✓ Dependiendo del alfabeto puede resultar difícil dividir una palabra en sus símbolos.
  - ✓ Si se puede dividir todas las palabras sobre un alfabeto solamente de una manera en sus símbolos, se llama tal alfabeto libre.
  - ✓ Solemos usar solamente alfabetos libres.
  - ✓ $|\varepsilon| = 0$

- El conjunto de todas las palabras que se pueden formar sobre un alfabeto $\sum$ más la palabra vacía se llama el **universo del alfabeto** $W(\sum)$.
  - ✓ $W(\sum) = \{\varepsilon\} \cup \{w \mid w \text{ es palabra sobre } \sum\}$
  - ✓ $\sum \subset W(\sum)$
  - ✓ ε es palabra de cualquier universo, $\varepsilon \in W(\sum)$.
  - ✓ La cardinalidad del universo es infinito (pero contable o enumerable, vemos más adelante lo que significa).
  - ✓ Si el alfabeto es libre (o mejor decir, un generador libre), escribimos $\sum^*$ por $W(\sum)$.

- Podemos **concatenar** palabras, entonces sean w, v y u palabras en $\sum^*$.
  - ✓ w.v = wv, es decir, usamos el punto(.) como símbolo de concatenación, pero muchas veces obviamos de él (igual como se suele hacer con el · de la multiplicación).
  - ✓ $\varepsilon w = w = w\varepsilon$, es decir, ε se comporta como el elemento neutro (o elemento de intentidad) respecto a la concatenación.
  - ✓ |w.v| = |w| + |v|
  - ✓ w.v ≠ v.w para cualquier w y v, por ejemplo:

    w = abc v = dec wv = abcdec ≠ decabc = vw

    es decir, la concatenación no es conmutativa.
  - ✓ (w.v).u = w.(v.u) para cualquier palabras w, v y u, por ejemplo:

    w = abc v = dec u = fad

    (wv)u = (abcdec)fad = abcdecfad = abc(decfad) = w(vu)

    es decir, la concatenación es asociativa (usamos arriba las paréntesis como metasímbolos).
  - ✓ Con dichas propiedades la estructura algebraica (__, .) forma un monoide libre (es decir, un semigrupo con elemento de intentidad).

- Si xy = w, llamamos x **prefijo** de w e y **sufijo** de w.
  - ✓ Por $\varepsilon w = w$ y $w\varepsilon = w$, ε es por lo tanto prefijo y sufijo (trivial) de cualquier palabra, y w es prefijo y sufijo trivial de sí mismo. (Normalmente no consideramos estos casos triviales.)
  - ✓ Si x es prefijo de w entonces $|x| \leq |w|$.
  - ✓ Si y es sufijo de w entonces $|y| \leq |w|$.
  - ✓ Si x es prefijo de w, e y es sufijo de w y x = y, entonces x = y = w

- Si concatenamos siempre la misma palabra w, obtenemos **potencias** de w.
  - ✓ ww = w2, www = w3
  - ✓ w1 = w, w0 = ε
  - ✓ |wi| = i · |w|
  - ✓ $|w0| = |\varepsilon| = 0 = 0 \cdot |w| = |w0|$
  - ✓ wm+n = wm.wn
  - ✓ |wm+n| = (m + n) · |w| = m · |w| + n · |w| = |wm| + |wn|

- La **reflexión** de una palabra w (o la palabra reversa) anotamos como w^R.
  - ✓ |w| = |wR|
  - ✓ ε = εR

5.1.1.3. OPERACIONES CON LENGUAJES.

Sean L, L1, L2, L3 ⊂ ∑* lenguajes (igual para W(∑)):

- **Unión**

  L1 ∪ L2 = {w | w ∈ L1 o w ∈ L2}

  - ✓ Propiedades (unos ejemplos):

    Conmutatividad: L1 ∪ L2 = L2 ∪ L1

    Asociatividad: (L1 ∪ L2) ∪ L3 = L1 ∪ (L2 ∪ L3)

    Idempotencia: L ∪ L = L

    Operación con ϕ: L ∪ ϕ = L = ϕ ∪ L

    Operación con ∑*: L ∪ ∑* = ∑* = ∑* ∪ L

- **Intersección:**

  L1 ∩ L2 = {w | w ∈ L1 y w ∈ L2}

  - ✓ Propiedades (unos ejemplos):

    Conmutatividad: L1 ∩ L2 = L2 ∩ L1

    Asociatividad: (L1 ∩ L2) ∩ L3 = L1 ∩ (L2 ∩ L3)

    Idempotencia: L ∩ L = L

    Operación con ϕ: L ∩ ϕ = ϕ = ϕ ∩ L1

    Operación con ∑*: L ∩ ∑* = L = ∑* ∩ L

- **Complemento:**

  $\overline{L} = \{w \mid w \in \sum^* \text{ y } w \notin L\}$

  - ✓ Propiedades (unos ejemplos):

    Regla de DeMorgan:

$$\overline{L_1 \cup L_2} = \overline{L}_1 \cap \overline{L}_2$$
$$\overline{L_1 \cap L_2} = \overline{L}_1 \cup \overline{L}_2$$

Con estas tres operaciones la estructura ($\Sigma^*, \cap, \cup,$) forma una álgebra booleana.

- **Diferencia:**

  L1 − L2 = {w | w ∈ L1 pero w ∉ L2}

  ✓ Propiedades (unos ejemplos):

$$\overline{L}_1 = \Sigma^* - L_1$$
$$L_1 - L_2 = L_1 \cap \overline{\Sigma^* \cap L_2)}$$

- **Concatenación:**

  L1.L2 = {w | w = w1.w2 y w1 ∈ L1 y w2 ∈ L2}

  ✓ Propiedades (unos ejemplos):

  No–Conmutatividad: L1.L2 ≠ L2.L1 (en general)

  Operación con ϕ: L1.ϕ = L1 = ϕ.L1

  Operación con {ε}: L1.{ε} = L1 = {ε}.L1

- **Potencia:**

$$L^i = \underbrace{L \ldots L}_{i\text{-veces}} \quad i \in \mathbb{N}$$

  ✓ Propiedades (unos ejemplos):

  Cero–Potencia: L0 = {ε}

  Inducción: Li.L = Li+1 = L.Li

- **Clausura positiva:**

$$L^+ = \bigcup_{i=1}^{\infty} L^i = L^1 \cup L^2 \cup L^3 \cup \ldots$$

- **Clausura (de Kleene):**

$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L^1 \cup L^2 \cup \ldots$$

- **Reflexión (o inverso):**

$L = \{w \mid w^R \in L\}$

5.1.2. LENGUAJES REGULARES.

Un lenguaje regular es un tipo de lenguaje formal que satisface las siguientes propiedades:

Puede ser reconocido por:

- Un autómata finito determinista
- Un autómata finito no determinista
- Un autómata finito alterno
- Una máquina de Turing de solo lectura

Es generado por:

- Una gramática regular
- Una gramática de prefijos

Es descrito por:

- Una expresión regular

5.1.2.1. LENGUAJES SOBRE ALFABETOS.

Un lenguaje regular sobre un alfabeto Σ dado se define recursivamente como:

- El lenguaje vacío $\varnothing$ es un lenguaje regular
- El lenguaje cadena vacía {ε} es un lenguaje regular
- Para todo símbolo a ∈ Σ {a} es un lenguaje regular
- Si A y B son lenguajes regulares entonces $A \cup B$ (unión), $A \bullet B$ (concatenación) y A^* (clausura o estrella de Kleene) son lenguajes regulares
- Si A es un lenguaje regular entonces (A) es el mismo lenguaje regular
- No existen más lenguajes regulares sobre Σ

Todo lenguaje formal finito constituye un lenguaje regular. Otros ejemplos típicos son todas las cadenas sobre el alfabeto {a, b} que contienen un

número par de a's o el lenguaje que consiste en varias a's seguidas de varias b's.

Si un lenguaje no es regular requiere una máquina con al menos una complejidad de Ω(log log n) (donde n es el tamaño de la entrada). En la práctica la mayoría de los problemas no regulares son resueltos con una complejidad logarítmica.

Un lenguaje formal infinito puede ser regular o no regular. El lenguaje $L = \{a^n, n > 0\}$ es regular porque puede ser representado, por ejemplo, mediante la expresión regular a^+. El lenguaje $L = \{a^n b^n, n > 0\}$ es un lenguaje no regular dado que no es reconocido por ninguna de las formas de representación anteriormente enumeradas[13].

5.1.2.2. EXPRESIONES REGULARES.

Una expresión regular, a menudo llamada también patrón, es una expresión que describe un conjunto de cadenas sin enumerar sus elementos. Por ejemplo, el grupo formado por las cadenas *Handel*, *Händel* y *Haendel* se describe mediante el patrón "H(a|ä|ae)ndel". La mayoría de las formalizaciones proporcionan los siguientes constructores: una expresión regular es una forma de representar a los lenguajes regulares (finitos o infinitos) y se construye utilizando caracteres del alfabeto sobre el cual se define el lenguaje. Específicamente, las expresiones regulares se construyen utilizando los operadores unión, concatenación y clausura de Kleene.

Alternación

Una barra vertical separa las alternativas. Por ejemplo, "marrón|castaño" casa con *marrón* o *castaño*.

[13] **Sánchez Isabel, Cárdenas María Antonia**, Teoría de autómatas y lenguajes formales, **URL:** http://www.yakiboo.net/apuntes/TALF%20-%20YakiBoo.net.pdf**.**

Cuantificación

Un cuantificador tras un carácter especifica la frecuencia con la que éste puede ocurrir. Los cuantificadores más comunes son +, ? y *:

+

El signo más indica que el carácter al que sigue debe aparecer al menos una vez. Por ejemplo, "ho+la" describe el conjunto infinito *hola*, *hoola*, *hooola*, *hoooola*, etcétera.

?

El signo de interrogación indica que el carácter al que sigue puede aparecer como mucho una vez. Por ejemplo, "ob?scuro" casa con *oscuro* y *obscuro*.

*****

El asterisco indica que el carácter al que sigue puede aparecer cero, una, o más veces. Por ejemplo, "0*42" casa con *42*, *042*, *0042*, *00042*, etcétera.

Agrupación

Los paréntesis pueden usarse para definir el ámbito y precedencia de los demás operadores. Por ejemplo, "(p|m)adre" es lo mismo que "padre|madre", y "(des)?amor" casa con *amor* y con *desamor*.

Los constructores pueden combinarse libremente dentro de la misma expresión, por lo que "H(ae?|ä)ndel" equivale a "H(a|ae|ä)ndel".

5.1.2.3. AFD.

Un AFD o autómata finito determinista es aquel en cual para cada par (estado, símbolo) está perfectamente definido el siguiente estado al cual pasará el autómata, es decir para cualquier estado q en que se encuentre el autómata y con cualquier símbolo s del alfabeto leído, existe exactamente una transición que parte de q y está etiquetada con s.

Formalmente, un autómata finito determinista (AFD) es similar a un Autómata de estados finitos, representado con una 5-tupla (S,Σ,T,s,A) donde:

1. Σ es un alfabeto;
2. *S* un conjunto de estados;
3. *T* es la función de transición: T: S x $\sum \rightarrow$ S
4. s $\in$ S es el estado inicial;
5. A $\subseteq$ S es un conjunto de estados de aceptación o finales.

Al contrario de la definición de autómata finito, este es un caso particular donde no se permiten transiciones vacías, el dominio de la función T es S (con lo cual no se permiten transiciones desde un estado de un mismo símbolo a varios estados).

A partir de este autómata finito es posible hallar la expresión regular resolviendo un sistema de ecuaciones.

$S_1 = 1\ S_1 + 0\ S_2 + \varepsilon$

$S_2 = 1\ S_2 + 0\ S_1$

Siendo ε la palabra nula. Resolviendo el sistema y haciendo uso de las reducciones apropiadas se obtiene la siguiente expresión regular: 1*(01*01*)*.

5.1.2.4. AFN.

Un AFND o autómata finito no determinista es aquel que presenta cero, una o más transiciones por el mismo carácter del alfabeto.

Un autómata finito no determinista también puede o no tener más de un nodo inicial.

Los *AFND* también se representan formalmente como tuplas de 5 elementos (*S*,Σ,*T*,*s*,*A*). La única diferencia respecto al AFD es T.

AFD: T: S x $\sum \rightarrow$ S

AFND: T: S x $\sum \rightarrow$ P(S) (partes de S)

Debido a que la función de transición lleva a un conjunto de estados, el autómata puede estar en varios estados a la vez (o en ninguno si se trata del conjunto vacío de estados).

5.1.2.5. TRANSICIONES.

En Teoría de autómatas y Lógica secuencial, una **tabla de transición de estados** es una tabla que muestra que estado (o estados en el caso de un Autómata finito no determinista) se moverá la máquina de estados, basándose en el estado actual y otras entradas. Una *tabla de estados* es esencialmente una tabla de verdad en la cual algunas de las entradas son el estado actual, y las salidas incluyen el siguiente estado, junto con otras salidas.

Tablas de estados de una dimensión

También llamadas **tablas características**, las tablas de estados de una dimensión son más como tablas de verdad que como las versiones de dos dimensiones. Las entradas son normalmente colocadas a la izquierda, y separadas de las salidas, las cuales están a la derecha. Las salidas representarán el siguiente estado de la máquina. Aquí hay un ejemplo sencillo de una máquina de estados con dos estados, y dos entradas combinacionales:

A	B	Estado Actual	Siguiente Estado	Salida
0	0	S_1	S_2	1
0	0	S_2	S_1	0
0	1	S_1	S_2	0
0	1	S_2	S_2	1
1	0	S_1	S_1	1
1	0	S_2	S_1	1
1	1	S_1	S_1	1
1	1	S_2	S_2	0

Tabla 01. Tabla de estados de una dimensión

S_1 y S_2 representarían probablemente los bits individuales 0 y 1, dado que un simple bit solo tiene dos estados[14].

5.1.2.6. PROPIEDADES DE LOS LENGUAJES REGULARES.

Existen diferentes herramientas que se pueden utilizar sobre los lenguajes regulares:

• El lema de Pumping: cualquier lenguaje regular satisface el pumping lemma, el cual se puede usar para probar que un lenguaje no es regular.

• Propiedades de cerradura: se pueden construir autómatas a partir de componentes usando operaciones, v.g., dado un lenguaje L y M construir un autómata para L ∩M.

• Propiedades de decisión: análisis computacional de autómatas, v.g., Probar si dos autómatas son equivalentes.

• Técnicas de minimización: útiles para construir máquinas más pequeñas.

La clase de lenguajes conocidos como lenguajes regulares tienen al menos 4 descripciones: DFA, NFA, ε − NFA y RE.

5.1.3. LENGUAJES INDEPENDIENTES DEL CONTEXTO.

5.1.3.1. GRAMÁTICAS REGULARES.

En informática una gramática regular es una gramática formal (N, Σ, P, S) que puede ser clasificada como regular izquierda o regular derecha. Las gramáticas regulares sólo pueden generar a los lenguajes regulares de manera similar a los autómatas finitos y las expresiones regulares.

Dos gramáticas regulares que generan el mismo lenguaje regular se denominan equivalentes. Toda gramática regular es una gramática libre de contexto.

Una gramática regular derecha es aquella cuyas reglas de producción P son de la siguiente forma:

[14] **Sánchez Isabel, Cárdenas María Antonia**, Teoría de autómatas y lenguajes formales, **URL:** http://www.yakiboo.net/apuntes/TALF%20-%20YakiBoo.net.pdf**.**

1. $A \rightarrow a$, donde A es un símbolo no-terminal en N y a uno terminal en Σ
2. $A \rightarrow aB$, donde A y B pertenecen a N y a pertenece a Σ
3. $A \rightarrow \varepsilon$, donde A pertenece a N.

Análogamente, en una gramática regular izquierda, las reglas son de la siguiente forma:

- $A \rightarrow a$, donde A es un símbolo no-terminal en N y a uno terminal en Σ
- $A \rightarrow Ba$, donde A y B pertenecen a N y a pertenece a Σ
- $A \rightarrow \varepsilon$, donde A pertenece a N.

5.1.3.2. GRAMÁTICAS INDEPENDIENTES DEL CONTEXTO.

Así como cualquier gramática formal, una gramática libre de contexto puede ser definida mediante la 4-tupla:

$G = (V_t, V_n, P, S)$ donde:

- V_t es un conjunto finito de terminales
- V_n es un conjunto finito de no terminales
- P es un conjunto finito de producciones
- $S \in V_n$ el denominado Símbolo Inicial
- los elementos de P son de la forma $V_n \rightarrow (V_t \cup V_n)^*$ [15]

5.1.3.3. ÁRBOLES DE DERIVACIÓN.

Un árbol de derivación permite mostrar gráficamente cómo se puede derivar cualquier cadena de un lenguaje a partir del símbolo distinguido de una gramática que genera ese lenguaje.

Un árbol es un conjunto de puntos, llamados nodos, unidos por líneas, llamadas arcos. Un arco conecta dos nodos distintos. Para ser un árbol un conjunto de nodos y arcos debe satisfacer ciertas propiedades:

[15] **Depto. de Informática Universidad de Valladoil**, Teoría de autómatas y lenguajes formales, 2009 **URL:** http://www.infor.uva.es/~teodoro/Ejer3WEB.pdf

- Hay un único nodo distinguido, llamado raíz (se dibuja en la parte superior) que no tiene arcos incidentes.
- Todo nodo c excepto el nodo raíz está conectado con un arco a otro nodo k, llamado el padre de c (c es el hijo de k). El padre de un nodo, se dibuja por encima del nodo.
- Todos los nodos están conectados al nodo raíz mediante un único camino.
- Los nodos que no tienen hijos se denominan hojas, el resto de los nodos se denominan nodos interiores.

El árbol de derivación tiene las siguientes propiedades:

- El nodo raíz está rotulado con el símbolo distinguido de la gramática;
- Cada hoja corresponde a un símbolo terminal o un símbolo no terminal;
- Cada nodo interior corresponde a un símbolo no terminal.

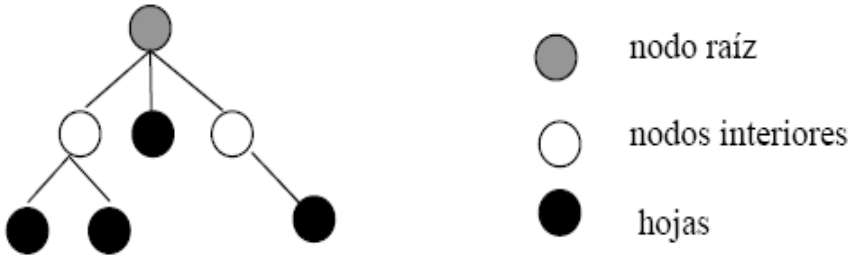


Figura 01. Arboles de derivación

5.1.3.4. SIMPLIFICACIÓN DE GRAMÁTICAS INDEPENDIENTES DEL CONTEXTO

Sea G = (V, T, P, S) una GIC
Un **símbolo** x ∈ (V U T) se dice que es **útil** si aparece durante el proceso de derivación que genera alguna palabra de L(G).

Formalmente:
$S \Rightarrow_G \alpha x \beta \Rightarrow_G \omega$; $\alpha, \beta \in (V \cup T)^*$, $\omega \in T^*$
Condiciones necesarias para que x sea útil:

1. Alguna cadena de terminales se deriva de x (**vivo**)
2. x aparece en alguna cadena derivada de S (**accesible**)

.

Eliminación de símbolos no vivos ó muertos

Sea G = (V, T, P, S) una GIC
W1 = {A ∈ V / (A → ω) ∈ P; ω ∈ T*}...
Wk+1 = Wk U {A ∈ V / (A → α) ∈ P; α ∈ (Wk U T)*}
W1 ⊆ W2 ⊆ … ⊆ Wk ⊆ Wk+1 ⊆ …
Existe n t.q. Wn = Wn+1 = Símbolos vivos
G' =(Wn, T, P', S) sin símbolos muertos
P' = {(A → α) ∈ P / A ∈ Wn, α ∈ (Wn U T)* }
Si S ∉ Wn entonces P' = ∅
L(G) = L(G')

Eliminación de símbolos inaccesibles

Sea G = (V, T, P, S) una GIC
W1 = {S}...
Wk+1 = Wk U {x ∈ (V U T) / ∃ α, β ∈ (V U T)*; B ∈ Wk con (B → α x β) ∈ P}
W1 ⊆ W2 ⊆ … ⊆ Wk ⊆ Wk+1 ⊆ …
Existe n t.q. Wn = Wn+1 = Símbolos accesibles
G' =(V ∩ Wn, T ∩ Wn , P', S) sin símbolos inaccesibles
P' = {(A → α) ∈ P / A ∈ V ∩ Wn}
L(G) = L(G')

Eliminación de Λ-producciones de GIC

Sea G = (V, T, P, S) una GIC
Una L-producción es una producción de la forma A → Λ
Una **variable** X se dice que es **anulable** si de ella puede derivarse Λ. Formalmente:
X ⇒*G Λ
Cálculo de anulables:

W1 = {A ∈ V / (A → Λ) ∈ P}...
Wk+1 = Wk U {A ∈ V / (A → α) ∈ P; α ∈ Wk*}
Existe n t.q. Wn = Wn+1 = Símbolos anulables
G' =(V, T, P', S) sin Λ-producciones
P' = P – {X → Λ / X ∈ V} U
{(B → αβ) / (B → α A β) ∈ P, A ∈ Wn, αβ ≠Λ}
Si S ∈ Wn entonces P' = P' U {S → Λ }[16]

5.1.4. ANÁLISIS LÉXICO

Analizador léxico (scanner): lee la secuencia de caracteres del programa fuente, carácter a carácter, y los agrupa para formar unidades con significado propio, *los componentes léxicos* (*tokens* en ingles). Estos componentes léxicos representan:

palabras reservadas: if, while, do, . . .
identificadores: asociados a variables, nombres de funciones, tipos definidos por el usuario, etiquetas,... Por ejemplo: posicion, velocidad, tiempo, . . .
operadores: = * + - / == > < & ! = . . .
símbolos especiales: ; ( ) [ ] *f g* ...
constantes numéricas: literales que representan valores enteros, en coma flotante, etc, 982, 0xF678, -83.2E+2,...
constantes de caracteres: literales que representan cadenas concretas de caracteres, \hola mundo",...

El analizador léxico opera bajo petición del analizador sintáctico devolviendo un componente léxico conforme el analizador sintáctico lo va necesitando para avanzar en la gramática. Los componentes léxicos son los símbolos terminales de la gramática.
Suele implementarse como una subrutina del analizador sintáctico. Cuando recibe la orden obtén el siguiente componente léxico,

[16] **Morales Luna Guillermo**, Árboles de Derivación, 2000 **URL:** http://delta.cs.cinvestav.mx/~gmorales/ta/node104.html

el analizador léxico lee los caracteres de entrada hasta identificar el siguiente componente léxico.

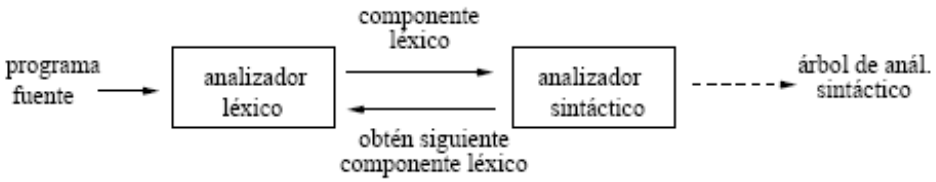


Figura 02. Analizador Léxico

Patrón: es una regla que genera la secuencia de caracteres que puede representar a un determinado componente léxico (una expresión regular).

Lexema: cadena de caracteres que concuerda con un patrón que describe un componente léxico.

Componente léxico: puede tener uno o infinitos lexemas. Por ejemplo: palabras reservadas tienen un único lexema. Los números y los identificadores tienen infinitos lexemas.

Los componentes léxicos se suelen definir como un tipo enumerado. Se codifican como enteros. También se suele almacenar la cadena de caracteres que se acaba de reconocer (el lexema), que se usaría posteriormente para el análisis semántico. [17]

5.2. COMPILADORES

5.2.1. ANÁLISIS SINTÁCTICO

Un **analizador sintáctico** (en inglés *parser*) es una de las partes de un compilador que transforma su entrada en un árbol de derivación.

El análisis sintáctico convierte el texto de entrada en otras estructuras (comúnmente árboles), que son más útiles para el posterior análisis y capturan la jerarquía implícita de la entrada. Un analizador léxico crea tokens de una secuencia de caracteres de entrada y son estos tokens los que son procesados por el analizador sintáctico para construir la estructura de datos, por ejemplo un árbol de análisis o árboles de sintaxis abstracta.

[17] Louden, K.C, Construcción de Compiladores: *Principios y Práctica*, 1997, Tema 2, páginas: 31-93.

El análisis sintáctico también es un estado inicial del análisis de frases de lenguaje natural. Es usado para generar diagramas de lenguajes que usan flexión gramatical, como los idiomas romances o el latín. Los lenguajes habitualmente reconocidos por los analizadores sintácticos son los lenguajes libres de contexto. Cabe notar que existe una justificación formal que establece que los lenguajes libres de contexto son aquellos reconocibles por un autómata de pila, de modo que todo analizador sintáctico que reconozca un lenguaje libre de contexto es equivalente en capacidad computacional a un autómata de pila.

5.2.2. ANÁLISIS SEMÁNTICO Y CÓDIGO INTERMEDIO

Se compone de un conjunto de rutinas independientes, llamadas por los analizadores morfológico y sintáctico.

El análisis semántico utiliza como entrada el árbol sintáctico detectado por el análisis sintáctico para comprobar restricciones de tipo y otras limitaciones semánticas y preparar la generación de código.

En compiladores de un solo paso, las llamadas a las rutinas semánticas se realizan directamente desde el analizador sintáctico y son dichas rutinas las que llaman al generador de código. El instrumento más utilizado para conseguirlo es la gramática de atributos.

En compiladores de dos o más pasos, el análisis semántico se realiza independientemente de la generación de código, pasándose información a través de un archivo intermedio, que normalmente contiene información sobre el árbol sintáctico en forma linealizada (para facilitar su manejo y hacer posible su almacenamiento en memoria auxiliar).

En cualquier caso, las rutinas semánticas suelen hacer uso de una pila (la pila semántica) que contiene la información semántica asociada a los operandos (y a veces a los operadores) en forma de *registros semánticos*.

5.2.3. GENERACIÓN DE CÓDIGO

El analizador sintáctico va generando acciones que valida el analizador semántico y que se convierten en tercetos. Esta conversión en tercetos constituye el generador de código intermedio.

Dado que el lenguaje puede presentar distintas funciones anidadas, los tercetos los generamos por orden del parser y son almacenados en un sitio u otro dependiendo del contexto en que nos encontremos. Es decir, se almacenan en una lista de tercetos dependiente de la Tabla de Símbolos. Hay tantas listas de tercetos como funciones haya en el código fuente más una lista de tercetos asociada a la Tabla de Símbolos Global

No obstante una vez finalizado el análisis, todos estos tercetos repartidos en distintas listas se vuelcan a una sola lista de tercetos global. Esta será la que finalmente se optimice y a partir de la que se generará el programa en ensamblador.

El problema de tener que manejar tercetos indirectos fue resuelto modificando el método de inserción sobre la lista de tercetos utilizada en cada momento, de manera que se realiza previamente una búsqueda de algún terceto que sea exactamente igual al que estamos insertando. En caso afirmativo, insertamos en la lista no un terceto nuevo, sino un puntero al ya existente, y marcamos dicho terceto como terceto indirecto. Son tercetos indirectos aquellos **marcados con un asterisco** después del índice en los volcados de la lista de tercetos.

5.2.4. OPTIMIZACIÓN Y GENERACIÓN DE CÓDIGO FINAL

En la etapa de **optimización de código**, se busca obtener el código más corto y rápido posible, utilizando distintos algoritmos de optimización.

Etapa de **generación de código,** final se lleva el código intermedio final a código maquina o código objeto, que por lo general consiste en un código maquina relocalizable o código ensamblador. Se selecciona las posiciones de memoria para los datos (variables) y se traduce cada una de las instrucciones intermedias a una secuencia de instrucciones de maquina puro. [18]

[18] Louden, K.C, Construcción de Compiladores*: Principios y Práctica*, 1997, Tema 2.

6. METODOLOGÍA

6.1. MÉTODOS

- **Método Inductivo:** Proceso que parte del estudio de casos o hechos singulares para llegar a los principios generales, este método lo aplicamos para poder determinar el problema partir de la información recolectada.
- **Método Deductivo:** Parte de lo general es decir lo conocido para inferir en las consecuencias particulares, aplicamos este método ya que a partir del problema determinamos las causas que lo originan para poder darle soluciones.

6.2. TECNICAS

- **Entrevista estructurada:** Entrevista en la cual se utiliza un cuestionario con preguntas elaboradas y ordenadas con relación al objetivo de la investigación, estas son planteadas por el investigador para que responda el encuestado, en este caso utilizamos esta entrevista para conocer cuáles son los problemas que tienen los alumnos de cuarto y quinto módulo de la carrera de Ingeniería en Sistemas, a quienes les realizamos la entrevista.

6.3. MÉTODOLOGÍA PARA DESARROLLO DEL SOFTWARE

Para obtener un software de calidad se ha considerado seguir un proceso adecuado haciendo uso de una metodología que mejor se ajuste a nuestro proyecto, es por ello que se ha realizado un respectivo análisis de las siguientes metodologías:

- XP (Extreme programming).
- RUP (Racional UnifiedProcess).
- ICONIX

Metodología	Fortaleza	Defectos
RUP	✓Demasiado soporte. ✓Utilizado y difundido en la industria. ✓Casos de uso y arquitectura ✓Se emplea para proyectos de Medio / Largo ✓El equipo de trabajo es de Medio / Largo ✓Se utiliza para problemas con complejidad de Media / Alta.	✓Complejidad de adaptación. ✓No hay guías disponibles para esto. ✓Posee mucha documentación.
ICONIX	✓Fácil de adaptar e implementar. ✓Conjunto pequeño de prácticas. ✓Iterativo e incremental. ✓Usa lo más importante de UML. ✓El equipo de trabajo es de Pequeño / Medio ✓Se utiliza para problemas con complejidad de Pequeña / Media.	✓No considera aspectos de manejo. ✓Se emplea para proyectos de Pequeños y medianos.
XP	✓Practicas basadas en valores simples y fáciles de transmitir. ✓Buen ambiente de trabajo. ✓Se puede combinar con otros enfoques ágiles. ✓El equipo de trabajo es Pequeño ✓Se utiliza para problemas con complejidad de Media / Alta.	✓Manejo sin mucho detalle. ✓Problemas en proyectos grandes. ✓Se requiere un alto esfuerzo por parte de los desarrolladores. ✓Se emplea para proyectos de Pequeño / Medio.

De acuerdo al análisis se ha determinado que la metodología a utilizar es XP ya que es un proyecto de complejidad media y el equipo de trabajo es pequeño, por lo tanto creemos que ésta metodología es la indicada debido a que se adapta correctamente a lo que necesitamos para realizar nuestro proyecto.

7. PRESUPUESTO Y FINANCIAMIENTO

7.1. Recursos Humanos

Recursos Humanos	Cantidad	Horas	Costo por Hora	Costo Total
Tesistas	2	800	$3	$2400
Director de Tesis	1	0	0	0
Subtotal				$ 2400

7.2. Recursos Técnicos y Tecnológicos

Recursos Técnicos	Cantidad	Horas	Costo por hora	Costo Total
Hardware				
Computadores	2	400	0.30	$120
Impresora HP D1560	1	10	0.50	$5
Software				
NetBeans IDE 6.0	2		Gratuito	$0
Eclipse Europa 3.3.5	2		Gratuito	$0
Java JDK 1.6	2		Gratuito	$0
Enterprise Architect 3.6	2		$ 335	$670
Microsoft Project	2		$170	$340
Paquete Microsoft Office	2		---	
	Subtotal			$ 1135

7.3. Recursos Materiales

Recursos Materiales	Cantidad	Costo Unitario	Costo Total
Resma de Papel	2	$4	$8
Bolígrafos	3	$0.2	$0.6
Cartuchos de tinta.	2	$20	$40
Internet/horas	50	$0.7	$35
Subtotal			$ 83.60

7.4. Gastos Generales

Recursos Materiales	Horas	Costo por Hora	Costo Total
Luz	800	$ 0.50	$ 400
Teléfono	300	$ 1	$300
Transporte	1000	$ 0.25	$ 250
Subtotal			$ 950

7.5. Total de Recursos

Resumen del Presupuesto	Costo Total
Recursos Humanos	$2400
Recursos Materiales	$83.60
Recursos Técnicos Tecnológicos	$1135
Gastos Generales	$950
SUBTOTAL	$4568.60
Imprevistos 10 %	$456.86
TOTAL	$5025.46

El costo de la tesis será asumido en su totalidad por los tesistas, tal y como se ha especificado en la justificación económica.

8. CRONOGRAMA

9. BIBLIOGRAFÍA

9.1. TEXTOS BASICOS

- ✓ Aho, Alfred V.; Ullman, Jeffrey D.; COMPILADORES: PRINCIPIOS, TÉCNICAS Y HERRAMIENTAS.
- ✓ Alfonseca Moreno, Manuel; Echeandía, Marina de la Cruz, Ortega de la Puente, Alfonso; Pulido Cañabate, Estrella. COMPILADORES E INTERPRETES: teoría y práctica, Pearson Prentice Hall, 2006.
- ✓ Castro Esteban, David. TEORÍA DE AUTÓMATAS, LENGUAJES FORMALES Y GRAMÁTICAS. Universidad de Alcalá. 2003-2004.
- ✓ Louden, Kenneth C. CONSTRUCCIÓN DE COMPILADORES PRINCIPIOS Y PRÁCTICA. International Thomson Editores. UNED. 2004.
- ✓ Louden, K.C, CONSTRUCCIÓN DE COMPILADORES*: Principlos y Práctica*, 1997, Tema 2
- ✓ Louden, K.C, CONSTRUCCIÓN DE COMPILADORES: PRINCIPLOS Y PRÁCTICA, 1997, Tema 2, páginas: 31-93.
- ✓ Luengo, Cándida, ANÁLISIS SINTÁCTICO EN PROCESADORES DE LENGUAJE, Oviedo, Enero 2005.
- ✓ Llano Diaz, Emiliano. ANÁLISIS Y DISEÑO DE COMPILADORES. México. Primera Edición. 2002.
- ✓ Martínez Gloria, García Luis A. APUNTES DE TEORÍA DE AUTÓMATAS Y LENGUAJES FORMALES. 2005.
- ✓ PARR, Terence. A quick tour for the impatient, THE DEFINITIVE ANTLR REFERENCES, BUILDING DOMAIN-SPECIFIV LANGUAGE, Dallas, Texas, pp. 59-84. 2010.
- ✓ Rojas Gálvez, Sergio; Mora Mata, Miguel Ángel. COMPILADORES: TRADUCTORES Y COMPILADORES CON LEX/YACC, JFLEX/CUP Y JAVACC. Universidad de Málaga. 2005.

9.2. TEXTOS ELECTRÓNICOS

- ✓ Morales Luna Guillermo, Árboles de Derivación, 2000 URL: [http://delta.cs.cinvestav.mx/~gmorales/ta/node104.html]
- ✓ Autómatas Finitos [http://es.wikipedia.org/w/index.php?title=Aut%C3%B3mata_finito&amp]
- ✓ Lenguaje regular [http://es.wikipedia.org/w/index.php?title=Lenguaje_regular&amp]
- ✓ Gramática Libre de Contexto [http://es.wikipedia.org/w/index.php?title=Gram%C3%A1tica_libre_de_contexto&amp]
- ✓ Ramón Brena. AUTÓMATAS Y LENGUAJES. Tecnológico de Monterrey. 2003. [http://homepages.mty.itesm.mx/rbrena/AyL.html].
- ✓ Depto. de Informática Universidad de Valladoil, Teoría de autómatas y lenguajes formales, 2009 URL: [http://www.infor.uva.es/~teodoro/Ejer3WEB.pdf].
- ✓ Sánchez Isabel, Cárdenas María Antonia, Teoría de autómatas y lenguajes formales, URL: [http://www.yakiboo.net/apuntes/TALF%20-%20YakiBoo.net.pdf].

10. ANEXOS

Matriz de Consistencia General

PROBLEMA GENERAL DE LA INVESTIGACIÓN: **"La falta de una buena metodología de aprendizaje de elaboración de algoritmos de los alumnos de cuarto y quinto modulo de la carrera de Ingeniería en Sistemas de la Universidad Nacional de Loja"**			
TEMA	**OBJETO DE LA INVESTIGACIÓN**	**OBJETIVO DE LA INVESTIGACIÓN**	**HIPOTESIS DE LA INVESTIGACION**
"IMPLEMENTACIÓN DE UN COMPILADOR EN ESPAÑOL PARA LA EJECUCIÓN DE ALGORITMOS EN PSEUDOCÓDIGO"	Compilador en español para la ejecución de algoritmos en pseudocódigo	Implementar un Compilador en español para la ejecución de algoritmos en pseudocódigo utilizando el framework ANTLR.	El uso de un compilador para la ejecución de algoritmos en pseudocódigo ayudará a mejorar y agilizar el proceso de enseñanza-aprendizaje de los alumnos de Cuarto y Quinto Módulo de la Carrera de Ingeniería en Sistemas de la Universidad Nacional de Loja.

MATRIZ DE CONSISTENCIA ESPECÍFICA (1)

OBTETIVO ESPECÍFICO: Realizar un analizador léxico, para comprobar que los lexemas y los tokens estén escritos correctamente		
PROBLEMA	**UNIDAD DE OBSERVACIÓN**	**SISTEMA CATEGORIAL**
✓ **El alfabeto no se ha definido completamente.** ✓ **No se ha determinado completamente los lexemas y los tokens.**	✓ Cuartos y Quintos Módulos de la Carrera de Ingeniería en Sistemas de la Universidad Nacional de Loja. ✓ Compilador	✓ Framework ANTLR ✓ Analizador Léxico

MATRIZ DE OPERATIVIDAD DE OBJETIVOS (1)

OBTETIVO ESPECÍFICO: Realizar un analizador léxico, para comprobar que los lexemas y los tokens estén escritos correctamente						
ACTIVIDAD O TAREA	**METODOLOGÍA**	**FECHA**		**RESPONSABLES**	**PRESUPUESTO**	**RESULTADOS ESPERADOS**
		INICIO	**FINAL**			
✓ Realizar el analizador léxico. ✓ Definir el alfabeto. ✓ Definir el lenguaje. ✓ Definir los lexemas. ✓ Definir los tokens.	✓ Framework ANTLR	13/10/10	26/11/10	Alondra Ordoñez Alex Román	25 25 25 25 25	✓ Analizador Léxico o Alfabeto o Lenguaje o Lexemas o Tokens

MATRIZ DE CONSISTENCIA ESPECÍFICA (2)

OBTETIVO ESPECÍFICO: Realizar un analizador sintáctico, que verifique que las sentencias estén escritas con la sintaxis correcta		
PROBLEMA	**UNIDAD DE OBSERVACIÓN**	**SISTEMA CATEGORIAL**
✓ **La gramática es ambigua.** ✓ **Las estructuras de datos y bucles están mal definidas.** ✓ **El árbol sintáctico se genera de forma incorrecta.**	✓ Cuartos y Quintos Módulos de la Carrera de Ingeniería en Sistemas de la Universidad Nacional de Loja. ✓ Compilador	✓ Framework ANTLR ✓ Analizador Sintáctico

MATRIZ DE OPERATIVIDAD DE OBJETIVOS (2)

OBTETIVO ESPECÍFICO: Realizar un analizador sintáctico, que verifique que las sentencias estén escritas con la sintaxis correcta						
ACTIVIDAD O TAREA	**METODOLOGÍA**	**FECHA**		**RESPONSABLES**	**PRESUPUESTO**	**RESULTADOS ESPERADOS**
		INICIO	**FINAL**			
✓ Construir el analizador sintáctico. ✓ Construir la gramática.	✓ Framework ANTLR	29/11/10	21/01/10	Alondra Ordoñez Alex Román	50 50	✓ Analizador Sintáctico. ✓ Gramática

MATRIZ DE CONSISTENCIA ESPECÍFICA (3)

OBTETIVO ESPECÍFICO: Realizar un analizador semántico, para determinar que las sentencias estén de acuerdo a las reglas del lenguaje.		
PROBLEMA	**UNIDAD DE OBSERVACIÓN**	**SISTEMA CATEGORIAL**
✓ **No se ha establecido exactamente la compatibilidad entre tipos de datos.** ✓ **No se detectan a tiempo los errores semánticos en las sentencias.**	✓ Cuartos y Quintos Módulos de la Carrera de Ingeniería en Sistemas de la Universidad Nacional de Loja. ✓ Compilador	✓ Framework ANTLR ✓ Analizador Semántico

MATRIZ DE OPERATIVIDAD DE OBJETIVOS (3)

OBTETIVO ESPECÍFICO: Realizar un analizador semántico, para determinar que las sentencias estén de acuerdo a las reglas del lenguaje.						
ACTIVIDAD O TAREA	**METODOLOGÍA**	**FECHA**		**RESPONSABLES**	**PRESUPUESTO**	**RESULTADOS ESPERADOS**
		INICIO	**FINAL**			
✓ Construir el Analizador Semántico.	✓ Framework ANTLR	24/01/11	04/03/10	Alondra Ordoñez Alex Román	100	✓ Analizador Semántico

MATRIZ DE CONSISTENCIA ESPECÍFICA (4)

OBTETIVO ESPECÍFICO: Realizar la generación de código intermedio para la ejecución de algoritmos en pseudocódigo.		
PROBLEMA	**UNIDAD DE OBSERVACIÓN**	**SISTEMA CATEGORIAL**
✓ **Dificultad al capturar el archivo fuente para convertirlo al lenguaje java.**	✓ Cuartos y Quintos Módulos de la Carrera de Ingeniería en Sistemas de la Universidad Nacional de Loja. ✓ Compilador	✓ Framework ANTLR ✓ Generación de código intermedio.

MATRIZ DE OPERATIVIDAD DE OBJETIVOS (4)

OBTETIVO ESPECÍFICO: Realizar la generación de código intermedio para la ejecución de algoritmos en pseudocódigo.						
ACTIVIDAD O TAREA	**METODOLOGÍA**	**FECHA**		**RESPONSABLES**	**PRESUPUESTO**	**RESULTADOS ESPERADOS**
		INICIO	**FINAL**			
✓ Generación de código intermedio.	✓ Framework ANTLR	07/03/11	30/05/11	Alondra Ordoñez Alex Román	300	✓ Generación de Código intermedio.

MATRIZ DE CONSISTENCIA ESPECÍFICA (5)

OBTETIVO ESPECÍFICO: Realizar las pruebas de validación del software		
PROBLEMA	**UNIDAD DE OBSERVACIÓN**	**SISTEMA CATEGORIAL**
✓ **La interfaz es demasiado compleja o el diseño no es el adecuado.** ✓ **El compilador no cumple con todos los requerimientos que el usuario pidió.**	✓ Cuartos y Quintos Módulos de la Carrera de Ingeniería en Sistemas de la Universidad Nacional de Loja. ✓ Compilador	✓ Evaluación y Pruebas de un Software.

MATRIZ DE OPERATIVIDAD DE OBJETIVOS (5)

OBTETIVO ESPECÍFICO: Realizar las pruebas de validación del software						
ACTIVIDAD O TAREA	**METODOLOGÍA**	**FECHA**		**RESPONSABLES**	**PRESUPUESTO**	**RESULTADOS ESPERADOS**
		INICIO	**FINAL**			
✓ Realizar la evaluación del Software con los alumnos de la Carrera de Ingeniería en Sistema y el docente. ✓ Corrección de errores.	✓ Entrevista con los alumnos y el docente.	01/06/11	30/06/11	Alondra Ordoñez Alex Román	50 50	✓ Listado de errores de la aplicación. ✓ Compilador en español para ejecución de algoritmos en pseudocódigo completamente acorde con las necesidades de los usuarios.

MATRIZ DE CONSISTENCIA ESPECÍFICA (6)

OBTETIVO ESPECÍFICO: Capacitación a estudiantes y docentes de la Carrera de Ingeniería en Sistemas		
PROBLEMA	**UNIDAD DE OBSERVACIÓN**	**SISTEMA CATEGORIAL**
✓ Falta de conocimiento sobre el funcionamiento de la aplicación. ✓ Falta de experiencia en el uso de herramientas de software de este tipo.	✓ Compilador	✓ Manual de Usuario ✓ Manual de Programador

MATRIZ DE OPERATIVIDAD DE OBJETIVOS (6)

OBTETIVO ESPECÍFICO: Capacitación a estudiantes y docentes de la Carrera de Ingeniería en Sistemas						
ACTIVIDAD O TAREA	**METODOLOGÍA**	**FECHA**		**RESPONSABLES**	**PRESUPUESTO**	**RESULTADOS ESPERADOS**
		INICIO	**FINAL**			
✓ Capacitar a los docentes de cuarto y quinto modulo de Ingeniería en Sistemas. ✓ Capacitar a los estudiantes de cuarto y quinto modulo de Ingeniería en Sistemas.	✓ Capacitación a los usuarios	01/07/11	15/07/11	Alondra Ordoñez Alex Román	20 20	✓ Manual de Usuario ✓ Manual del Programador.

MATRIZ DE CONTROL DE RESULTADOS

No.	RESULTADOS	FECHA DE INICIO	FECHA DE ENTREGA	FIRMA DEL DOCENTE
	ANALISIS Y DISEÑO DEL SISTEMA			
1	Determinar los Requerimientos del sistema	01/09/10	07/09/10	
2	Elaborar el glosario de términos	08/09/10	10/09/10	
3	Elaborar el modelo de dominio y el diagrama de clases	13/09/10	24/09/10	
4	Elaborar el diagrama de casos de uso	27/09/10	01/10/10	
5	Elaborar el prototipo de pantallas	05/10/10	11/10/10	
	CODIFICACIÓN			
6	Construir el Analizador Léxico	13/10/10	12/11/10	
7	Determinar Lexemas	15/11/10	19/11/10	
8	Determinar Tokens	22/11/10	26/11/10	
9	Construir el Analizador Sintáctico	29/11/10	21/01/11	
10	Construir el Analizador Semántico	24/01/11	04/03/11	
11	Generar Código Intermedio	07/03/11	30/05/11	
	EVALUACIÓN DEL SOFTWARE			
12	Corrección de errores	01/06/11	30/06/11	
	CAPACITACIÓN A LOS USUARIOS			
13	Capacitación a alumnos y docente	01/07/11	15/07/11	

2.8.5. Anexo C