Instituto Tecnológico de Costa Rica

Ingeniería en Computación

Compiladores e Intérpretes

Proyecto 1

Profesor: Ing. Allan Rodríguez Dávila

Estudiante:	Carnet:
Jordano Escalante	2018161994

Segundo Semestre

Fecha: 06 de octubre

Año:2025

Índice:

Contenido

Descripción del problema

Se requiere desarrollar una Gramática BNF para un lenguaje imperativo con las siguientes características:

a) Permitir la creación de funciones, y dentro de ellas, estructuras de control, bloques de código (¿ y ?) y sentencias de código.
b) Se permite crear variables globales.
c) Manejar los tipos de variables enteras, flotantes, booleanas, caracteres, cadenas de caracteres (string) y arreglo estático unidimensional.
d) Se permite crear arreglos de tipo char y entero. Además, se permite obtener y modificar sus elementos, y ser utilizados en expresiones. Se permite creación con asignación (¿ y ?).
e) Permitir sentencias para creación de variables (locales y globales), creación y asignación de expresiones y asignación de expresiones a variables, y algunos casos, sólo expresiones sin asignación. En el caso de creaciones utilizar la palabra reservada let.
f) Las expresiones permiten combinar literales, variables, arreglos y/o funciones, de los tipos reconocidos en la gramática.
g) Debe permitir operadores y operandos, respetando precedencia (usual matemática) y permitiendo el uso de paréntesis (є y ɜ).
h) Permitir expresiones aritméticas binarias de suma (+), resta (-), división (// y /) entera o decimal según el tipo--, multiplicación (*), módulo (%) y potencia (^). Para enteros o reales. Permitir expresiones aritméticas unarias de negativo (-), ++ y -- después del operando; el negativo se puede aplicar a literales enteros y flotantes, el ++ y -- se aplica a variables enteres y flotantes.
i) Permitir expresiones relacionales (sobre enteros y flotantes) de menor, menor o igual, mayor, mayor o igual, igual y diferente. Los operadores igual y diferente permiten adicionalmente tipo booleano.
j) Permitir expresiones lógicas de conjunción (@), disyunción (~) y negación (ésta debe ser de tipo caracter (Σ)).
k) Debe permitir sentencias de código para las diferentes expresiones mencionadas anteriormente y su combinación, el delimitador de final de expresión será el carácter dólar ($). Además, dichas expresiones pueden usarse en las condicionales y bloques de las siguientes estructuras de control.
l) Debe permitir el uso de tipos y la combinación de expresiones aritméticas (binarias y unarias), relacionales y lógicas, según las reglas gramaticales, aritméticas, relacionales y lógicas del Paradigma Imperativo, por ejemplo, tomando como referencia el lenguaje
m) Cualquier duda con respecto al comportamiento de la funcionalidad debe validarlo con el profesor.
n) La gramática genera un lenguaje con tipado explícito y fuerte.
o) Debe permitir las funciones de leer (enteros y flotantes) y escribir en la salida estándar (cadena carácter, enteros, boolean y flotantes), se pueden escribir literales o variables. Utilizar palabras reservadas output e input.
p) Debe permitir la creación y utilización de funciones, estos deben retornar valores (entero, flotantes, char y booleanos) y recibir parámetros (con tipo).

q) Debe definir un único procedimiento inicial main (llamado principal), por medio de la cual se inicia la ejecución de los programas, este es de tipo void y no recibe parámetros.
r) Además, debe permitir comentarios de una línea (|) o múltiples líneas (¡ y !).
s) Debe permitir las funciones de leer (enteros y flotantes) y escribir en la salida estándar (cadena carácter, enteros, boolean y flotantes), se pueden escribir literales o variables. Utilizar palabras reservadas output e input.
t) Debe permitir la creación y utilización de funciones, estos deben retornar valores (entero, flotantes, char y booleanos) y recibir parámetros (con tipo).
u) Debe definir un único procedimiento inicial main (llamado principal), por medio de la cual se inicia la ejecución de los programas, este es de tipo void y no recibe parámetros.
v) Debe permitir las estructuras de control:
decide of, de la forma:
decide of
(condicion -> bloque)*
(else -> bloque)?
end decide$

loop (tipo Ada, no usa llaves de bloque)
loop instrucciones…
exit when condicion$
end loop$

for (variación de Pascal con to y downto, utiliza llaves después del do para definir un bloque)
for asignacion step valor (to o downto) expresion do
bloque

Además, permitir return y break. Las expresiones de las condiciones deberán ser valores booleanos combinando expresiones aritméticas, lógicas y relacionales.

Diseño del programa

La gramática se desarrolló usando la notación Backus Naur estudiada en el curso.
Se siguió el siguiente estándar:
nombreElemento -> expresión/símbolos/otros para los no terminales, terminales, símbolo inicial y producciones, además se decidió evitar el uso de caracteres especiales propios del español ya que puede incluir problemas de notación que conviene evitar en el desarrollo de una gramática para mantenerla lo más sencilla posible.

La lista completa de terminales, no terminales y producciones se encuentra en el documento adjunto llamado “gramatica BNF.txt” que se encuentra en el repositorio del proyecto, pueden ser consultadas con mejor detalle en el mismo y además se encuentran agrupadas por su tipo y uso.

Manual de usuario:

Para generar los archivos necesarios para el funcionamiento del proyecto es necesario que utilice los archivos jar que se encuentran en el directorio “lib” del proyecto, asegúrese de contar con todos los archivos necesarios que se detallan a continuación:

- java-cup-11b-runtime.jar
- java-cup-11b.jar
- jflex-full-1.9.1.jar

Es probable que las versiones que descargue del sitio oficial de jflex y jcup no sean las mismas que las que se detallan anteriormente, en cuyo caso asegúrese de modificar los scripts de generación de lexer y parser adecuadamente.

Para generar el lexer y parser y posteriormente ejecutar el análisis sintáctico ejecute los siguientes comandos en la terminal en el orden que se indican.

Para ejecutar la generación del Lexer.java ejecute:
java -jar lib\jflex-full-1.9.1.jar src\lexer.flex
El archivo resultado Lexer.java debería aparecer en la carpeta “src”.

Para generar el parser ejecute:
java -jar lib\java-cup-11b.jar -destdir src -parser Parser src\sintax.cup
En la carpeta “src” deberían aparecer los archivos Parser.java y sym.java.

Compile los archivos java para generar las clases necesarias:
javac -cp "lib*" src*.java
Este comando debería generar todas las clases necesarias para el funcionamiento en el directorio “src”.

Finalmente, regrese a la carpeta raíz del proyecto y para realizar el análisis sintáctico ejecute el comando:
java -cp "programa\lib*;programa\src" Proye1_compi
Es indispensable que se ejecute este comando en la carpeta raíz del proyecto ya que de lo contrario las rutas de acceso a los archivos se repetirán y no funcionará debidamente.

Este último script tomará el contenido del archivo “test.txt” del directorio “input” y generará en el directorio “output” los archivos “TablaSimbolos.txt” y “TOKENS.txt” con el contenido correspondiente al código analizado del archivo “test.txt”.

Pruebas de funcionalidad:

Aseguramiento de poseer los archivos jar.

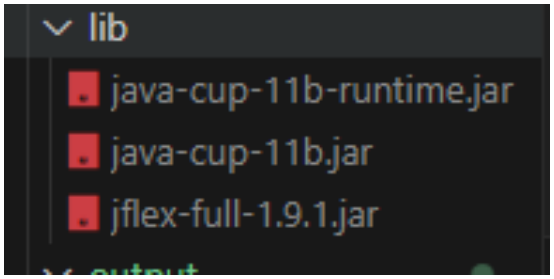

Imagen 1: jar files de cup y jflex.

Generación del lexer.

```
PS D:\Compilador2025\programa> java -jar lib\jflex-full-1.9.1.jar src\lexer.flex
Reading "src\lexer.flex"

Warning: Macro "true" has been declared but never used.

Warning: Macro "digitNoZero" has been declared but never used.

Warning: Macro "digit" has been declared but never used.
Constructing NFA : 467 states in NFA
Converting NFA to DFA :
.....................................................................................
.....................................................................................
222 states before minimization, 199 states in minimized DFA
Writing code to "src\Lexer.java"
```

Imagen 2: Ejecución del script para generar el Lexer.

El script genera el archivo java en la carpeta “src”.

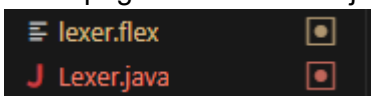

Imagen 3: Archivo Lexer.java generado correctamente el el directorio “src”.

Ejecución del comando para generar el parser el el directorio “src”.

```
PS D:\Compilador2025\programa> java -jar lib\java-cup-11b.jar -destdir src -parser Parser src\sintax.cup
Warning : LHS non terminal "error" has not been declared
Warning : Terminal "COMMENT" was declared but never used
Warning : Terminal "DOT" was declared but never used
Warning : Terminal "END_COMMENT" was declared but never used
Warning : Terminal "INIT_COMMENT" was declared but never used
Warning : Non terminal "decideOfBody" was declared but never used
Warning : Non terminal "expr" was declared but never used
------- CUP v0.11b 20160615 (GIT 4ac7450) Parser Generation Summary -------
  0 errors and 7 warnings
  67 terminals, 43 non-terminals, and 164 productions declared,
  producing 349 unique parse states.
  6 terminals declared but not used.
  0 non-terminals declared but not used.
  0 productions never reduced.
  0 conflicts detected (0 expected).
  Code written to "Parser.java", and "sym.java".
----------------------------------------------------- (CUP v0.11b 20160615 (GIT 4ac7450))
```

Imagen 4: Ejecución del comando para la generación del Parser y mensajes en consola al ejecutar.

La ejecución del comando generará el archivo Parser.java y sym.java.

Imagen 5: Archivo Parser.java generado en el directorio “src”.

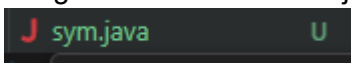

Imagen 6: Archivo sym.java generado en el directorio “src”.

Seguidamente se ejecuta el comando para la compilación y generación de clases necesarias para el funcionamiento del proyecto.

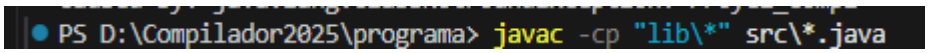

Imagen 7: Ejecución del script para la compilación del proyecto.

La ejecución del script anterior generará las clases Lexer.class, Parser.class, ParserCUPParser$actions.class, Proye1_compi.class y sym.class necesarias para el funcionamiento correcto del proyecto.

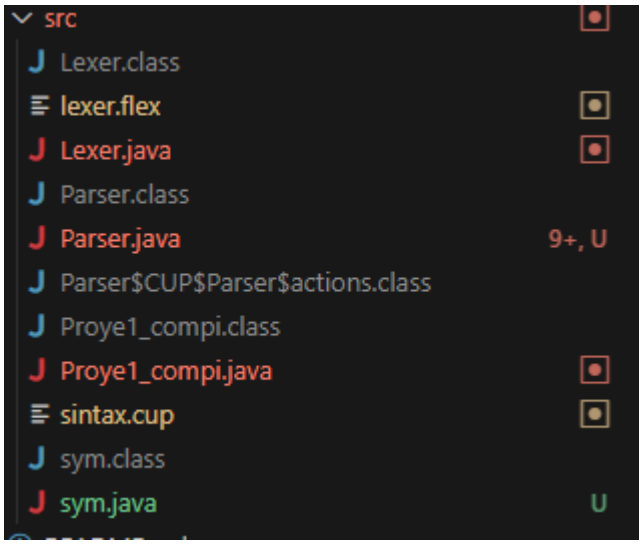

Imagen 8: Todos los .java y .class generados para el funcionamiento adecuado del proyecto.

Finalmente se ejecuta el comando para el análisis léxico y generación del archivo con los tokens y tablas de símbolos.

```
PS D:\Compilador2025> java -cp "programa\lib\*;programa\src" Proye1_compi
Token: 38 let (LET) at 1:1
Token: 31 int (INT) at 1:5
Token: 2 x (IDENTIFIER) at 1:9
Token: 66 $ (DOLLAR) at 1:10
Token: 37 global (GLOBAL) at 3:1
Token: 38 let (LET) at 3:8
Token: 31 int (INT) at 3:12
Token: 2 x (IDENTIFIER) at 3:16
Token: 21 = (EQ) at 3:17
Token: 11 1 (INT_LITERAL) at 3:18
Token: 66 $ (DOLLAR) at 3:19
Token: 38 let (LET) at 4:1
Token: 33 char (CHAR) at 4:5
Token: 2 i (IDENTIFIER) at 4:10
Token: 66 $ (DOLLAR) at 4:11
Cantidad de lexemas encontrados: 15
Tabla de simbolos escrita en: D:\Compilador2025\programa\output\TablaSimbolos.txt
El archivo se ha escrito correctamente en: D:\Compilador2025\programa\output\TOKENS.txt

TABLA DE SIMBOLOS (resumen)
Scope: SCOPE GLOBAL  (3 entradas)
Tablas de simbolos exportadas a: D:\Compilador2025\programa\output

TABLA DE SIMBOLOS (resumen)
Scope: SCOPE GLOBAL  (3 entradas)
Tablas de simbolos exportadas a: D:\Compilador2025\programa\output
```

Imagen 9: Ejecución del comando para el análisis léxico y mensajes de consola. El comando generaría los archivos “TOKENS.txt” y “TablasSimbolos.txt”.

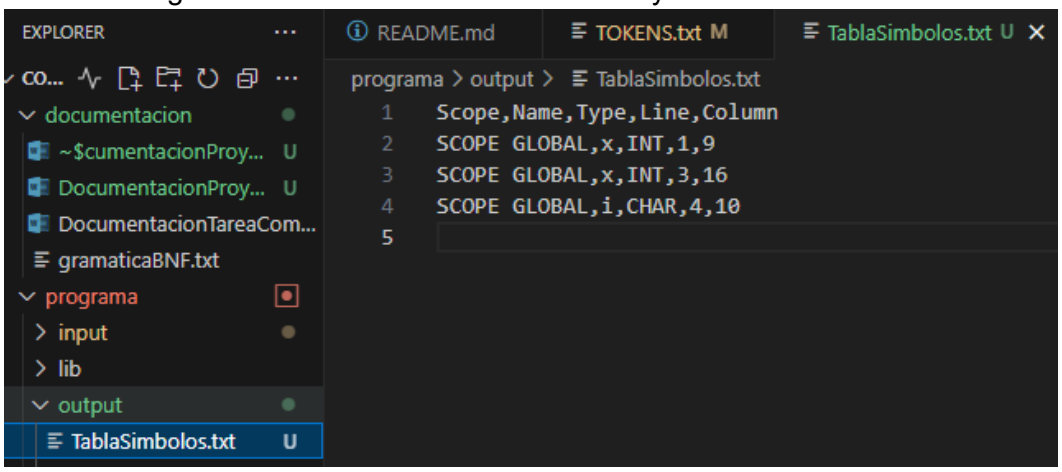

Imagen 10: Archivo “TablasSimbolos.txt” y su contenido generado.

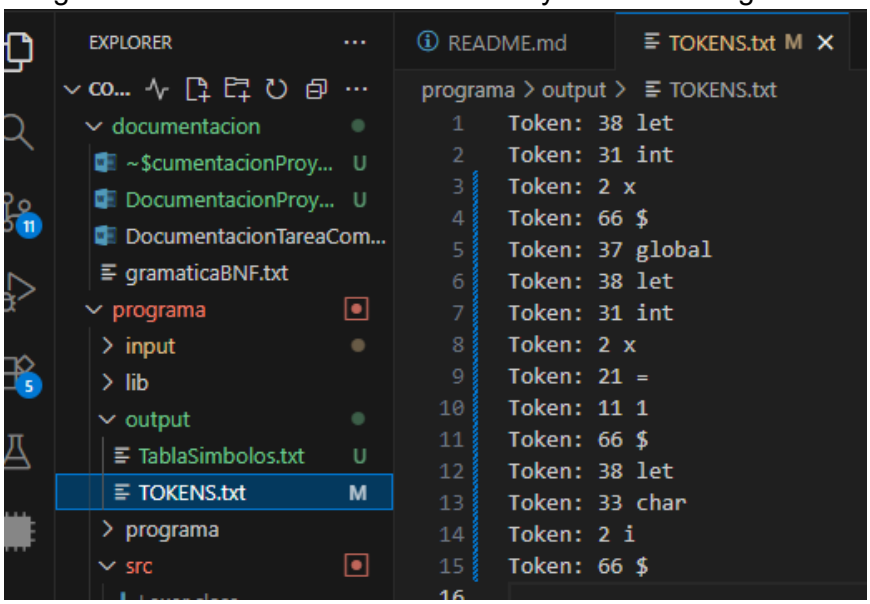

Imagen 11: Archivo “TOKENS.txt” y su contenido generado.

Diseño del programa:

De acuerdo con las especificaciones de jflex y jcup los archivos lexer.flex y sintax.cup son indispensables para la creación de proyectos utilizando estas dos herramientas de forma conjunta, por lo que para este proyecto se diseñó un proyecto con la distribución de archivos basada en la funcionalidad, por tanto, se dividió el mismo en 4 subdirectorios:

- Input: todos los archivos de prueba que el programa principal analizaría.
- Lib: Todos los jar necesarios para el funcionamiento de jflex y jcup.
- Output: carpeta donde se guardarán los datos de todos los tokens y tablas de símbolos.
- Src: carpeta donde se encuentran todas las clases, archivos java y archivos jflex y jcup.

Tanto las clases principales como archivos de flex y cup se mantuvieron en el mismo directorio para evitar problemas de enrutamiento.

En el archivo lexer.flex se encuentra la estructura basada en el archivo plantilla que se encuentra en el sitio oficial de la herramienta, solamente se agregó lógica necesaria para obtener la información de los símbolos que se procesan y se cambiaron los nombres de los símbolos y agregaron otros para satisfacer los requerimientos del proyecto.

El archivo sintax.cup posee la lista completa de terminales, no terminales y producciones necesarias para satisfacer los requerimientos de este proyecto, además posee la lógica para manejo de errores, impresión de tabla de símbolos, impresión de lista de tokens, validaciones del scope de cada iteración e impresiones en consola para mensajes claros.

La clase principal Proye1_compi contiene la lógica para utilizar las clases creadas mediante los scripts mencionados en la sección de manual de usuario y realizar así el análisis sintáctico.

La estructura general del proyecto se basa en las recomendaciones dadas en el sitio oficial de cup y flex y recomendaciones del profesor del curso.

Justificación de toma de decisiones:

Las producciones PROGRAMA y PRINCIPAL son las necesarias para la inicialización de la gramática y permiten que el código a analizar pueda derivarse de distintas formas, PROGRAMA funciona solamente como inicio y PRINCIPAL como un auxiliar que permite que la derivación tome distintas vías.

- BLOCK permite que dentro de un bloque de código pueda haber todo tipo de expresiones, por eso es la que tiene derivación hacia todo tipo de producciones.
- Numbers es la producción intermedia que permite definir dentro de las producciones que utilizan números cualquier número, sean enteros o flotantes.
- compAritOp es una simple comparación de mayor que o menor que entre dos términos.
- Term: puede ser cualquier termino en una operación aritmética.

- Factor puede ser cualquier elemento que pueda tomar forma de un factor en una operación, sea este aritmético o lógico.
- inputStruc hace referencia a los argumentos en una operación o función.
- exprP es una producción intermedia que permite definir que una expresión puede ser lógica o unitaria.
- opRel son operaciones relacionales.
- opLog son operaciones lógicas.
- vasAsig hace referencia a la asignación a una variable.
- vasIns es la producción que permite todos los tipos de instanciación de variables con todos los tipos de datos que una variable puede tener.
- vasInsGl igual que varIns pero globales.
- vasInsAsig y varInsAsigGl son instanciación y asignación de variable locales o globales.
- funInitial son las producciones para definición de funciones de distintos tipos de datos.
- funcStruct es la estructura para definición de funciones.
- paraFunc son los parámetros de funciones.
- funcInvo para invocación de funciones.
- mainInitial y mainStruct para iniciación de programa.
- decideOfStruct – decideOfElseStruct producciones necesarias para crear la estructura decideOf.
- LoopStruct y forStruct son las producciones que definen el comportamiento de ambas estructuras de control tal como decideOf…
- arrayIns-arrayAsig son las producciones para definir arreglos.
- Elementos son los elementos de un arreglo.
- paramStruct y paramList son las producciones para definir los distintos tipos de parámetros en el código.
- Error es la producción que arroja los distintos tipos de errores que pueden aparecer en la ejecución del análisis.

La precedencia que se definición fue basada en las reglas de gramática típicas conocidas, el uso de estructuras intermedias (ejemplo numbers) se hace con el fin de simplificar el cuerpo de estructuras grandes, por ejemplo:
En una producción donde uno de los no terminales puede ser tanto exprUni o exprLog, con solo definir que puede ser exprP ya la gramática se puede derivar ya que exprP es una expresión auxiliar que se define precisamente con el fin de lograr este comportamiento, así para cada expresión de este tipo definida en la gramática.

Análisis de resultados

Objetivo	Estado	Lección
Creación de funciones	80%	Tiene errores
Manejar tipos	80%	Tiene errores
Manejar Variables	80%	Tiene errores
Manejar expresiones	80%	Tiene errores
Creación de variables	80%	Tiene errores
Crear arreglos	80%	Tiene errores
Definir alcance de variables	80%	Tiene errores
Combinación de expresiones	80%	Tiene errores
Operadores y operandos	80%	Tiene errores
Operaciones aritméticas	80%	Tiene errores
Operaciones aritméticas Unarias	80%	Tiene errores
Expresiones relacionales	80%	Tiene errores
Expresiones lógicas	80%	Tiene errores
Tipado	80%	Tiene errores
Estructuras de control	80%	Tiene errores
Entrada y salida estándar	80%	Tiene errores
Procedimiento inicial main	80%	Tiene errores
Comentarios	80%	Tiene errores
Manejo de errores	50%	No se alcanzó con éxito que los mensajes fueran 100% descriptivos y siempre marca errores a pesar de que no existan.

Bitácora

La bitácora fue generada de manera automática usando el sistema de control de versiones Git, puede consultar el repositorio online de la herramienta accediendo al siguiente enlace: https://github.com/EscalanteWizard/Compilador2025.git