

Escuela de ingeniería en Computación

Compiladores e Intérpretes

*Parser*

Profesora:

Erika Marín Schumann

Alumnos:

Marvin J. Cortés Castillo

Joseph Ramírez Moreno

I Semestre 2016

Tabla de contenido

[Análisis de Resultados 3](#_Toc453180814)

[Aspectos relevantes 6](#_Toc453180815)

[Casos de prueba con resultados erróneos reparados 7](#_Toc453180816)

[Caso No.1 Estructura del Programa 7](#_Toc453180817)

[Resultados esperados 7](#_Toc453180818)

[Resultados obtenidos 7](#_Toc453180819)

[Caso No.2 Expresiones 7](#_Toc453180820)

[Resultados esperados 7](#_Toc453180821)

[Resultados obtenidos 7](#_Toc453180822)

[Caso No.3 Declaración de Variables 8](#_Toc453180823)

[Resultados esperados 8](#_Toc453180824)

[Resultados obtenidos 8](#_Toc453180825)

[Caso No.4 Funciones 8](#_Toc453180826)

[Resultados esperados 8](#_Toc453180827)

[Resultados obtenidos 8](#_Toc453180828)

[Caso No.5 Bloques de Código 8](#_Toc453180829)

[Resultados esperados 8](#_Toc453180830)

[Resultados obtenidos 8](#_Toc453180831)

[Caso No.6 Estructura de control IF 8](#_Toc453180832)

[Resultados esperados 9](#_Toc453180833)

[Resultados obtenidos 9](#_Toc453180834)

[Caso No.7 Estructura de control TRY 9](#_Toc453180835)

[Resultados esperados 9](#_Toc453180836)

[Resultados obtenidos 9](#_Toc453180837)

[Caso No.8 Estructura de control FOR y WHILE 9](#_Toc453180838)

[Resultados esperados 9](#_Toc453180839)

[Resultados obtenidos 9](#_Toc453180840)

[Caso No.9 Parámetros de Funciones, Print e Input 10](#_Toc453180841)

[Resultados esperados 10](#_Toc453180842)

[Resultados obtenidos 10](#_Toc453180843)

[Caso No.10 Expresiones de Lista y Tuplas 10](#_Toc453180844)

[Resultados esperados 10](#_Toc453180845)

[Resultados obtenidos 10](#_Toc453180846)

[Casos de prueba con resultados correctos 10](#_Toc453180847)

[Caso No.11 Evaluando funcionalidad completa 10](#_Toc453180848)

[Resultados esperados 11](#_Toc453180849)

[Resultados obtenidos 11](#_Toc453180850)

[Ejecución del programa 11](#_Toc453180851)

[Descarga del IDE 11](#_Toc453180852)

[Instalación 11](#_Toc453180853)

[Ejecución del programa 11](#_Toc453180854)

[Gramática del Programa 13](#_Toc453180855)

# Análisis de Resultados

|  |  |  |
| --- | --- | --- |
| **Requerimiento** | **Nivel de completitud** | **Comentario** |
| **Listado de errores léxicos encontrados**:  El programa  debe desplegar una lista de todos los errores  léxicos que se encontraron en el código fuente. Debe desplegar la línea en la que se encontró el error. Es importante que el programa  deba poder recuperarse del error y no desplegar los errores en cascada ni terminar de hacer el escaneo al encontrar el primer error. | 100% completado | El programa muestra los errores léxicos una vez encontrados |
| **Listado de errores sintácticos encontrados**:  El programa debe desplegar una lista de todos los  errores sintácticos que se encontraron en el código fuente. Debe desplegar la línea en la que se encontró  el error. Además, el mensaje de error debe ser lo más específico posible, con el fin de que el  programador pueda llegar al error y corregirlo de forma eficiente. Es importante que el programa deba recuperarse del error y evitar no desplegar errores en cascada ni terminar de hacer el parseo al encontrar el primer error. | 100% completado | El programa muestra los errores sintácticos una vez encontrados |
| Las palabras reservadas que se deben incluir  en la gramática son las siguientes:  ***and, break , class, continue, def, elif, else, except, finally, for, if, in, input , is not, or, print, return, try, while, int, list, float, string, boolean, char*** | 100% completado | Muchas de las palabras ya habían sido definidas en el proyecto anterior, por consecuencia el programa acepta una cantidad mayor a las que se especifican |
| La estructura del programa puede ser de cualquiera de las siguientes formas. | 100% completado | El programa fue realizado para que solo existiera una única clase (no puede haber más de una clase).  Por otra parte si pueden haber infinita cantidad de funciones, definición de variables y código principal. |
| Las variables pueden ser de tipo ***int, float, list, string, boolean*** y ***char***.  La declaración de variables es de la siguiente forma: **Tipo nombre** | 100% completado | Igualmente, el programa acepta la declaración de la siguiente forma:  **Tipo nombre1, nombre2, ..** |
| La estructura de las funciones es de la siguiente manera: | 100% completado |  |
| El parser debe validar todo tipo de expresiones aritméticas y booleanas. Recuerde que una llamada a función también se considera como una expresión. | 100% completado |  |
| Las asignaciones se hacen así:  **Variable = Expresion** | 100% completado |  |
| Definir la estructura para la función **print** e **input** | 100% completado | Se definió una versión especifica del input la cual es:  **Id = input (expr)**  **input (expr)** |
| Recuerde que una llamada a función puede ser del tipo: **Funcion()** , o puede ser una función de una clase: **clase.funcion() ,** También son válidas la creación de clases: **c = Clase()** | 100% completado |  |
| Las estructuras de control tienen la siguiente estructura:  **while condición : If condición:**  **bloque bloque**  **; ;**  **elif condición:**  **for i in range() : bloque**  **bloque ;**  **; else:**  **bloque**  **;** | 100% completado |  |
| En un bloque pueden venir: asignaciones, otras estructuras de control, llamadas a funciones o ***print*** o ***input***. Dentro de las estructuras de control de ciclo pueden venir las sentencias ***break*** y ***continue***. | 100% completado | Se realizó una distinción entre los bloques que están dentro de las estructuras de ciclo y las demás estructuras de control |
| Implementar la estructura para el control de los errores  **try:**  **bloque**  **;**  **except Identificador:**  **bloque**  **;**  **finally:**  **bloque** | 100% completado | En comparación con el enunciado, se les agrego ; a los bloques o estructuras que tuvieran : |
| Los operadores que se deben tomar en cuenta son los siguientes:  *ARITMETICOS:*  **+, -, \*, /, //, %, \*\*, =, +=, -, =, \*=, /=, \*\*=, //=, /=, (, )**  *BOOLEANOS:* **!=, <>, >, <, >=, <=, ==, AND, OR, NOT** | 100% completado | El programa acepta más operadores que los solicitados. |
| En este aspecto es importante recordar que las condiciones de las estructuras de control utilizan solamente expresiones  booleanas. Para las asignaciones si pueden tener ambas operaciones. | 100% completado |  |

# Aspectos relevantes

1. Fue necesaria implementar una gramática un poco más estructurada para aceptar las múltiples declaraciones de variables; por ejemplo: ***Int a,b,c,d*** .
2. Se tuvo que implementar un tipo distinto para los bloques dentro de las estructuras de ciclos; puesto que estos pueden tener ***break*** y ***continue,*** razón por la cual se implementó una regla extra de la gramática para poder cumplir con dicho requerimiento descrito.
3. En algunos casos fue importante cambiar el símbolo de recuperación de error de la gramática, puesto que en diversas ocasiones no se podía obtener el error emergente.
4. Se modificaron ciertas reglas léxicas del proyecto anterior con el fin de lograr una integración acorde con las necesidades presentes en el actual proyecto.
5. Fue importante implementar una estructura para el manejo y despliegue de los distintos tipos de errores léxicos y sintácticos que el programa manejará.
6. Se implementó las expresiones de listas, tupas y precedencias de las expresiones, esto con el objetivo de adelantar trabajo para el siguiente proyecto.

# Casos de prueba con resultados erróneos reparados

## Caso No.1 Estructura del Programa

Se ingresó al archivo de entrada del programa, un código con una estructura invalida (diferente a la que el programa debía de aceptar)

### Resultados esperados

El programa reconociera que el código ingresado era erróneo, presto a que estaba estructurado de una manera diferente a la que se debía de acepta como válido.

### Resultados obtenidos

El programa aceptó el código ingresado con errores de estructura y lo tomaba como válido.

## Caso No.2 Expresiones

Se ingresó al archivo de entrada del programa, expresiones válidas e inválidas para probar la funcionalidad de las reglas de la gramática que evalúan esta parte.

### Resultados esperados

Solamente fueran aceptadas aquellas expresiones que cumplían con los requerimientos del programa, además detectara errores en aquellas expresiones no válidas para el programa; mostrando la fila y columna en donde estas se encontraban en el código.

### Resultados obtenidos

El programa reconocía como válidos a ciertos tipos de expresiones, las cuales eran erróneas y no debía de aceptarse. Además, aquellas expresiones que detectaba como inválidas no mostraban su número de fila y columna correctamente

## Caso No.3 Declaración de Variables

Se ingresó al archivo de entrada del programa, declaración de variables múltiples para ser evaluadas; por ejemplo ***int a,b,c***

### Resultados esperados

El programa debía reconocer como válidas dichas declaraciones.

### Resultados obtenidos

El programa reconoció como errores sintácticos las declaraciones.

## Caso No.4 Funciones

Se ingresó al archivo de entrada del programa, múltiples tipos de funciones, válidas e invalidas, para probar el correcto funcionamiento de las reglas de la gramática.

### Resultados esperados

Se tomarán como inválidas aquellas funciones que contaban con errores tanto de sintaxis como léxicos. Por otra parte, que aceptaran aquellas funciones que eran válidas para el programa.

### Resultados obtenidos

El programa acepto ciertas funciones válidas, tomó como inválidas funciones que debía de aceptar. Además, no detectó errores en funciones inválidas.

## Caso No.5 Bloques de Código

Se ingresó al archivo de entrada del programa, asignaciones, estructuras de control, llamadas a funciones, print’s e inputs para garantizar su correcto funcionamiento

### Resultados esperados

El programa pudiera aceptar aquellas estructuras descritas correctamente, además, pudiera haber distintos tipos de éstas en el bloque. Por otra parte, debía rechazar aquellas que no estuvieran definidas para el segmento de bloque.

### Resultados obtenidos

El sistema aceptaba ciertos tipos de estructuras válidas del segmento de bloque, pero no permitía que hubiera combinaciones de éstas. Del mismo modo no detectaba como errores a elementos que no debían pertenecer al segmento de bloque.

## Caso No.6 Estructura de control IF

Se ingresó al archivo de entrada del programa, diversos tipos válidos e inválidos de “if” para demostrar la correcta funcionalidad de las reglas de la gramática.

### Resultados esperados

El programa aceptara aquellas estructuras libres de errores y válidas para lenguaje, por otra parte, aquellas estructuras que contaba con errores debían ser tomadas como inválidas y demostrar información del error.

### Resultados obtenidos

El sistema tomaba como errores las estructuras que contaban con múltiples instrucciones “elif”, además existían inconsistencias en las descripciones de los errores.

## Caso No.7 Estructura de control TRY

Se ingresó al archivo de entrada del programa, diversos tipos válidos e inválidos de “try” para demostrar la correcta funcionalidad de las reglas de la gramática.

### Resultados esperados

El programa aceptara aquellas estructuras libres de errores y válidas para lenguaje, por otra parte, aquellas estructuras que contaba con errores debían ser tomadas como inválidas y demostrar información del error.

### Resultados obtenidos

El sistema tomaba como errores las estructuras que no contaban con la instrucción “finally”, además existían inconsistencias en las descripciones de los errores.

## Caso No.8 Estructura de control FOR y WHILE

Se ingresó al archivo de entrada del programa, diversos tipos válidos e inválidos de “for” y “while”, además, se ingresaros segmentos de bloques con instrucciones “break” y “continue” para demostrar la correcta funcionalidad de las reglas de la gramática.

### Resultados esperados

El programa aceptara aquellas estructuras libres de errores y válidas para lenguaje, por otra parte, aquellas estructuras que contaba con errores debían ser tomadas como inválidas y demostrar información del error. De mismo modo debía tomar como válidas las instrucciones “break” y “continue” que se encontraban dentro de los ciclos

### Resultados obtenidos

El sistema tomaba como errores las estructuras que no contaban con las instrucciones “break” y “continue” dentro de los ciclos; del mismo modo aceptaba dichas instrucciones fuera de las estructuras de ciclo. Por otra parte, existían inconsistencias en las descripciones de los errores.

## Caso No.9 Parámetros de Funciones, Print e Input

Se ingresó al archivo de entrada del programa, diversos tipos válidos e inválidos de parámetros de funciones, if e input, para demostrar la correcta funcionalidad de las reglas de la gramática.

### Resultados esperados

El programa aceptara aquellos parámetros libres de errores y válidos para lenguaje, por otra parte, aquellos parámetros que contaban con errores debían ser tomados como inválidos y mostrar información del error.

### Resultados obtenidos

El sistema tomaba como correctos parámetros sin tipo para las funciones, para el print tomaba parámetros sin “,” y para el input tomaba como válidos una la lista de expresiones.

## Caso No.10 Expresiones de Lista y Tuplas

Se ingresó al archivo de entrada del programa, diversos tipos válidos e inválidos de listas y tuplas, para demostrar la correcta funcionalidad de las reglas de la gramática.

### Resultados esperados

El programa aceptara aquellos expresiones de listas, tuplas, y combinaciones entre ellas libres de errores y válidos para lenguaje.

### Resultados obtenidos

El sistema tomaba como inválidas las listas de distintos tipos de expresiones con tuplas.

# Casos de prueba con resultados correctos

## Caso No.11 Evaluando funcionalidad completa

Se ingresó al archivo de entrada del programa, múltiples elementos válidos e inválidos por un programa Python, para demostrar la correcta funcionalidad de todas las reglas de la gramática.

### Resultados esperados

Las estructuras, expresiones y tokens válidos fueran reconocidos y aceptados por el programa. Por otra parte, aquellas estructuras, expresiones y tokens no válidos fueran reconocidos, listados, ubicados e impresos para el usuario

### Resultados obtenidos

Los elementos válidos fueron reconocidos satisfactoriamente al igual que no los no válidos.

# Ejecución del programa

## Descarga del IDE

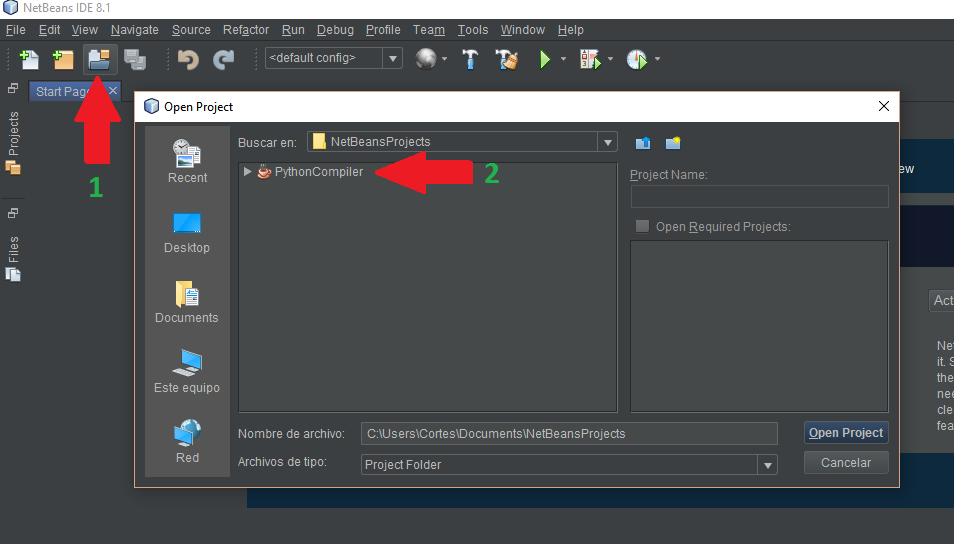
El programa fue desarrollado en NetBeans 8.1, el cual es un IDE multiplataforma de Java que facilita la creación y ejecución de proyectos en dicho lenguaje, en el siguiente link se puede acceder a descargar la versión más actualizada del IDE <https://netbeans.org/downloads/>

# Instalación

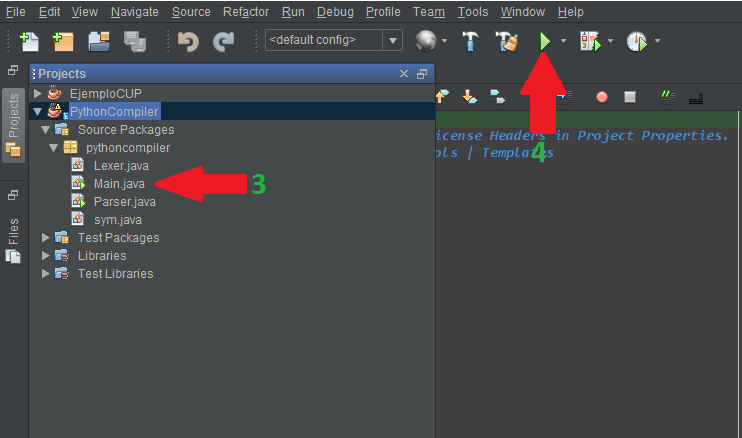
Para garantizar una segura y rápida instalación se recomienda seguir las instrucciones detalladas en el siguiente enlace <https://netbeans.org/community/releases/81/install.html>

## Ejecución del programa

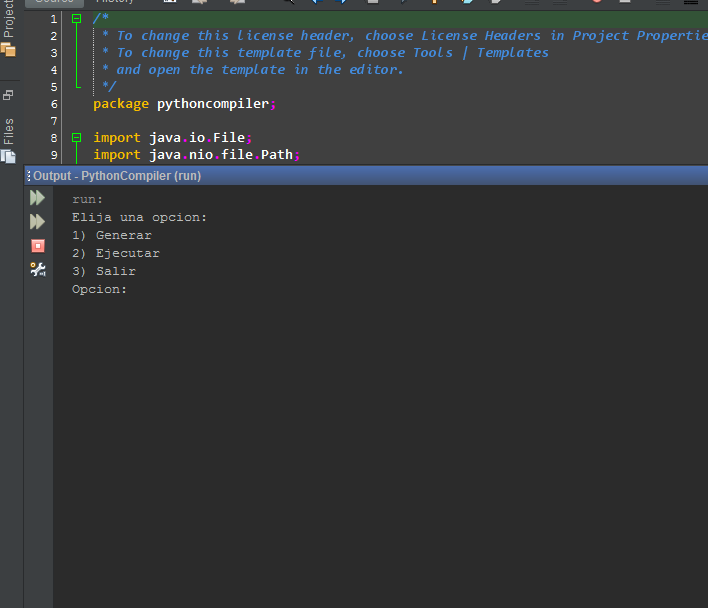
Una vez instalado el IDE procedemos a su ejecución. Primeramente, damos click en la opción de “abrir un proyecto existente”; tal y como se muestra en el punto #1 de la imagen. Seguidamente buscamos la ubicación de la carpeta del proyecto llamado “PythonCompiler”; como se muestra en el punto #2 de la imagen



Seguidamente ubicamos y ejecutamos el archivo Main del proyecto, dando click en el botón verde como se muestra a continuación en los puntos 3, y 4 de la imagen.



Una vez ejecutado el programa, desplegara un menú de consola el cual tiene las opciones de 1. Generar; el cual compila el proyecto y genera todas las declaraciones necesarias del programa, 2. Ejecutar; el cual realizará el parseo del código, en este punto es importante aclarar que el archivo de entrada se encuentra en la raíz del proyecto llamado “input.py”; y puede ser accedido a él mediante nuestro editor de texto preferido.



# Gramática del Programa

Seguidamente se muestran los símbolos terminales y no terminales del programa

**Terminales:** ERROR\_LIT, ERROR\_ID, ERROR\_INV, ERROR\_CHA, SEMICOLON\_OP, BIT\_OP, ARITH\_OP, L\_PARENTHESIS\_OP, R\_PARENTHESIS\_OP, NUM\_LIT, AND, BREAK, CLASS, CONTINUE, DEF, ELIF, ELSE, EXCEPT, FINALLY, FOR, IF, IN, INPUT, IS\_NOT, OR, PRINT, TRY, WHILE, TYPE\_LIST, TYPE\_INT, TYPE\_FLOAT, TYPE\_CHAR, TYPE\_STRING, TYPE\_BOOLEAN, COMP\_OP, ASSIG\_OP, LOGICAL\_OP, PROPERTY\_OP, COLON\_OP, IDENTIFIER, L\_BRACKET\_OP, R\_BRACKET\_OP, BOOLEAN\_LIT, CHAR\_LIT, STRING\_LIT, TAB\_OP, COMMA, RANGE, RETURN

**No-terminales:** program, def\_func\_stmt, fun\_stmt, fun\_stmt2, class\_stmt , list\_var\_decl, var\_decl, func\_par, block\_stmt, list\_block\_stmt, if\_stmt, elif\_stmt, else\_stmt, if\_loop\_stmt, elif\_loop\_stmt, else\_loop\_stmt, try\_stmt, except\_stmt, finally\_stmt, try\_loop\_stmt, except\_loop\_stmt, finally\_loop\_stmt, while\_stmt, for\_stmt, for\_param, logic\_expr, min\_expr, var\_assing, list\_var\_assig, call\_func\_expr, call\_func\_expr2, par\_expr, expr, arith\_expr, list\_arg, print\_stmt, input\_stmt, input\_stmt2, range\_stmt, range\_values, list\_loop\_block\_stmt, loop\_block\_stmt, base\_block\_stmt, bracket\_expr, return\_stmt ,type\_decl, id\_list, test1, test2, test3 ,beacket\_expr