Entrega 1

Curso: TC3002B.503

Estudiante:

Marco Ottavio Podesta Vezzali - A00833604

Fecha de entrega: 12/05/2025

Github repo: https://github.com/marcopod/compi/tree/main/entregable4

ANTLR (ANother Tool for Language Recognition) (Target Python)
https://www.antlr.org/

ANTLR es una herramienta generadora de analizadores que permite definir gramáticas en un formato declarativo basado en EBNF. Aunque está desarrollada en Java, permite generar código en múltiples lenguajes, incluido Python, facilitando la integración en proyectos Python sin necesidad de reescribir la funcionalidad del analizador.

Formato de entrada

- Las gramáticas se definen en archivos con extensión *.g4*.
- La sintaxis de estas gramáticas es declarativa, basándose en EBNF para especificar reglas léxicas y sintácticas.
- Las reglas para el análisis léxico se definen mediante patrones que utilizan expresiones regulares, mientras que las reglas sintácticas describen la estructura del lenguaje a analizar.
- Las reglas sintácticas se escriben debajo, con nombres en minúsculas, definiendo cómo los tokens se combinan para formar estructuras más complejas.

- Se pueden incluir **acciones semánticas** directamente en la gramática mediante bloques de código en el lenguaje de destino (Python en este caso).
-

Tipo de ejecución

- Para la generación de código, ANTLR toma el archivo .g4 y genera automáticamente el código fuente necesario (clases de lexer, parser, visitor, listener).
- Una vez generado, el analizador puede integrarse dentro de un proyecto Python para procesar entradas y generar árboles de sintaxis (parse trees) o recorrer el árbol utilizando visitors o listeners.

Técnicas de análisis

- Análisis léxico: Basado en expresiones regulares integradas en la gramática.
- Análisis sintáctico: Utiliza LL(*) ("LL-star"), una mejora de los parsers LL tradicionales, permitiendo predicción basada en la evaluación de múltiples tokens hacia adelante (lookahead) y backtracking opcional.

Tipo de licenciamiento

- ANTLR se distribuye bajo una licencia de código abierto similar a BSD o una licencia permisiva comparable, lo que permite su uso en proyectos tanto comerciales como de código abierto.

Herramientas teóricas empleadas

- Análisis léxico: Se apoya en expresiones regulares definidas directamente en la gramática para identificar tokens.
- Análisis sintáctico: Utiliza métodos de análisis LL(*) que permiten manejar un amplio rango de gramáticas y resolver ambigüedades mediante técnicas predictivas y, cuando es necesario, estrategias de backtracking.

Tipo de interfaz

- Las definiciones se realizan en archivos de texto (*.g4*) que especifican tanto los tokens (reglas léxicas) como la estructura (reglas sintácticas) de forma declarativa.
- No cuenta con una interfaz gráfica; la configuración se realiza mediante la edición del archivo de gramática, asistido por IDEs que suelen ofrecer realce sintáctico y validación.
- La salida generada es un conjunto de módulos Python que contienen clases y métodos para iniciar el análisis y manejar los árboles de sintaxis resultantes.

Facilidad para añadir código propio

- Dentro del archivo *.g4* es posible incrustar bloques de código en Python en secciones designadas (por ejemplo, en las secciones de acción), los cuales se ejecutan al reconocer una regla determinada.
- Se puede separar la definición de la gramática de la lógica de procesamiento adicional, permitiendo incluir funciones o métodos personalizados en el código Python generado o en módulos auxiliares que interactúan con el analizador.
- La generación del código de ANTLR permite modificar o ampliar el comportamiento del analizador sin alterar el proceso de compilación, facilitando la incorporación de validaciones o transformaciones adicionales.

Tecnológico de Monterrey

Lark https://github.com/lark-parser/lark

Lark es una librería para la generación de analizadores léxicos y sintácticos, completamente implementada en Python.

Formato de entrada

- Las gramáticas se definen utilizando una sintaxis basada en EBNF.
- Se pueden especificar las reglas tanto en cadenas de texto dentro del código como en archivos externos.
- Las reglas léxicas se establecen mediante expresiones regulares integradas en la definición de la gramática, y las reglas sintácticas describen la estructura jerárquica del lenguaje a analizar.

Tipo de ejecución

- LALR(1): Para gramáticas deterministas donde se busca mayor rendimiento.
- Earley: Para gramáticas ambiguas o recursivas, permitiendo manejar estructuras complejas sin la rigidez del análisis determinista.
- Al compilar la gramática, Lark genera de forma inmediata el árbol de derivación (parse tree) durante la ejecución en Python, sin etapas de compilación externa.

Tipo de licenciamiento

- Lark se distribuye bajo licencia MIT, lo que lo hace completamente gratuito para uso comercial y de código abierto, ofreciendo gran flexibilidad en la integración y distribución de proyectos.

Herramientas teóricas empleadas

- Análisis léxico: Se basa en expresiones regulares para reconocer tokens, integradas de forma nativa en la gramática.
- Análisis sintáctico: Implementa técnicas LALR(1) para gramáticas deterministas y el algoritmo Earley para aquellas ambiguas o con recursión profunda, proporcionando una solución adaptable al tipo de lenguaje y complejidad de la gramática.

Tipo de interfaz

- La gramática se define mediante cadenas de texto o archivos externos y se integra mediante una API Python sencilla.

- No posee una interfaz gráfica propia; la definición y configuración se hace directamente en el código, aprovechando la sencillez y flexibilidad de Python.
- La salida principal es la generación de árboles de análisis (parse trees), los cuales pueden ser recorridos, transformados y manipulados mediante callbacks o transformadores definidos por el usuario.

Facilidad para añadir código propio

Permite asociar funciones o transformadores a reglas específicas, mediante la integración de acciones semánticas.

Extensibilidad

- Es sencillo combinar la gramática con código Python adicional. Se pueden definir clases y funciones que procesen o modifiquen el árbol de análisis, integrándose de forma natural en el flujo del programa.
- La configuración y extensión del analizador no requieren pasos de compilación externos, permitiendo iteraciones rápidas durante el desarrollo.

Tecnológico de Monterrey

ANTLR es una herramienta madura y poderosa para la generación de analizadores léxicos y sintácticos, con un enfoque declarativo mediante archivos .g4 basados en EBNF. Aunque su núcleo está en Java (por lo que se necesita del JDK). Soporta múltiples lenguajes de destino, incluyendo Python. Destaca por su capacidad de integrar acciones semánticas en la gramática, su análisis sintáctico avanzado basado en LL(*) que permite lookahead múltiple y backtracking. Su enfoque requiere un proceso de compilación previo para generar las clases de lexer, parser, y visitors o listeners en Python.

Lark es una librería nativa de Python, más ligera y directa para construir analizadores, sin necesidad de procesos de generación de código externos. Soporta tanto análisis determinista (LALR(1)) como análisis más flexible y robusto (Earley), adaptándose bien a diferentes tipos de gramáticas, incluidas las ambiguas.

Resumen comparativo:

- ANTLR ofrece más potencia y opciones de optimización para proyectos grandes y complejos, pero hay una curva de configuración más pronunciada.
- Lark es nativo en python, por lo que resulta más fácil de implementar. Además, siendo este un proyecto relativamente sencillo, se puede implementar con Lark.

Tecnológico de Monterrey

I. Diseñar las Expresiones Regulares

Token	Descripción	Regex (estándar)
KW_PROGRAM	program	\bprogram\b
KW_VAR	var	\bvar\b
KW_MAIN	main	\bmain\b
KW_VOID	Void	\bvoid\b
KW_INT	Palabra reservada int	\bint\b
KW_FLOAT	Palabra reservada float	\bfloat\b
ID	Nombre (letra seguida de letras/dígitos/_)	[a-zA-Z_][a-zA-Z0-9_]*
KW_WHILE	while	\bwhile\b
KW_DO	do	\bdo\b
KW_END	end	\bend\b
KW_IF	if	\bif\b
KW_ELSE	else	\belse\b
KW_PRINT	print	\bprint\b
CTE_INT	Constante entera	[0-9]+
CTE_FLOAT	Constante flotante	[0-9]+\.[0-9]+
CTE_STRING	Cadena entre comillas	"([^"\\]|\\.)*"
PLUS	Operador suma	\+
MINUS	Operador resta	-
MULT	Operador multiplicación	*
DIV	Operador división	/
LT	Menor que	<
GT	Mayor que	>

NEQ	Diferente (!=)	!=
ASSIGN	Asignación (=)	=
SEMICOLON	Punto y coma	;
COMMA	Coma	,
LPAREN	Paréntesis izquierdo	\(
RPAREN	Paréntesis derecho	\)
LBRACE	Llave izquierda	{
RBRACE	Llave derecha	}
COLON	Dos puntos	:

II. Gramática libre de contexto

<PROGRAMA> ::= KW_PROGRAM ID SEMICOLON <VARS> <FUNCS> KW_MAIN <Body> KW_END
<VARS> ::= KW_VAR <ID_LIST> COLON <TYPE> SEMICOLON <VARS> | ε
<ID_LIST> ::= ID <ID_LIST_TAIL>
<ID_LIST_TAIL>::= COMMA ID <ID_LIST_TAIL> | ε
<TYPE> ::= KW_INT | KW_FLOAT
<CTE> ::= CTE_INT | CTE_FLOAT
<EXPRESIÓN> ::= <EXP>

Tecnológico de Monterrey

| <EXP> GT <EXP> | <EXP> LT <EXP> | <EXP> NEQ <EXP>
<EXP> ::= <TÉRMINO> <EXP'>
<EXP'> ::= PLUS <TÉRMINO> <EXP'> | MINUS <TÉRMINO> <EXP'> | ε
<TÉRMINO> ::= <FACTOR> <TÉRMINO'>
<TÉRMINO'> ::= MULT <FACTOR> <TÉRMINO'> | DIV <FACTOR> <TÉRMINO'> | ε
<FACTOR> ::= LPAREN <EXPRESIÓN> RPAREN | PLUS <FACTOR> | MINUS <FACTOR> | ID | <CTE>
<ASSIGN> ::= ID ASSIGN <EXPRESIÓN> SEMICOLON
<Print> ::= KW_PRINT LPAREN <PrintList> RPAREN SEMICOLON
<PrintList> ::= <PrintItem> <PrintListTail>
<PrintListTail> ::= COMMA <PrintItem> <PrintListTail> | ε
<PrintItem> ::= <EXPRESIÓN> | CTE_STRING
<F_Call> ::= ID LPAREN <ArgList> RPAREN SEMICOLON
<ArgList> ::= <EXPRESIÓN> <ArgListTail>
<ArgListTail> ::= COMMA <EXPRESIÓN> <ArgListTail> | ε

Tecnológico de Monterrey

<STATEMENT> ::= <ASSIGN> | <CONDITION> | <CYCLE> | <F_Call> | <Print>
<BODY> ::= LBRACE <STATEMENT_LIST> RBRACE
<STATEMENT_LIST> ::= <STATEMENT> <STATEMENT_LIST> | ε
<CONDITION> ::= KW_IF LPAREN <EXPRESIÓN> RPAREN <BODY> <ELSE_OPT> SEMICOLON
<ELSE_OPT> ::= KW_ELSE <BODY> | ε
<CYCLE> ::= KW_WHILE LPAREN <EXPRESIÓN> RPAREN KW_DO <BODY> SEMICOLON
<FUNCS> ::= <FUNC> <FUNCS> | ε
<FUNC> ::= KW_VOID ID LPAREN <PARAM_LIST_OPT> RPAREN <VARS> <Body> SEMICOLON
<PARAM_LIST_OPT> ::= <PARAM_LIST> | ε
<PARAM_LIST> ::= ID COLON <TYPE> <PARAM_LIST_TAIL>
<PARAM_LIST_TAIL> ::= COMMA ID COLON <TYPE> <PARAM_LIST_TAIL> | ε

Tecnológico de Monterrey

Test Plan para Compiler

1. Pruebas de Programas Completos

Caso	Descripción	Código de prueba	Resultado esperado
1.1	Programa mínimo válido	program a; main {} end	AST con programa vacío.
1.2	Programa con variables y cuerpo vacío	program b; var x: int; main {} end	AST donde vars no esté vacío.

2. Pruebas de Declaraciones de Variables (vars)

Caso	Descripción	Código de prueba	Resultado esperado
2.1	Variables tipo int	var x, y: int;	Lista de variables con tipo int.
2.2	Variables tipo float	var a: float;	Lista de variables con tipo float.
2.3	Sin variables	*(no se escribe var)*	vars vacío en AST.

3. Pruebas de Statements

Tipo de statement	Código ejemplo	Resultado esperado
Asignación	x = 5;	Nodo assign con ID y valor 5.

Llamada a función	foo(1, 2);	Nodo call con argumentos.
Condicional if-else	if (x != y) { } else { } ;	Nodo if con neq y rama else.
Ciclo while	while (x > 0) do { } ;	Nodo while con gt.

4. Pruebas de Expresiones (expr, term, factor)

Expresión	Código	Resultado esperado
Aritmética básica	x + y;	Nodo add.
Aritmética anidada	(x - 2) * 3;	Nodos sub, luego mult.

Tecnológico de Monterrey

Estructuras agregadas

1. **Tabla de Variables (VariableTable)**

Estructura usada: dict[str, VariableEntry]

Se eligió un diccionario porque permite acceder, insertar y verificar variables en tiempo constante. Esto es especialmente útil en un compilador donde la verificación de declaraciones duplicadas debe ser rápida y eficiente. Además, el modelo de un solo ámbito (por función o global) se adapta perfectamente a una tabla plana como esta, sin necesidad de estructuras jerárquicas más complejas.

Operaciones principales:

- add_variable(name, tipo, address)
- has_variable(name)
- get_variable(name)
- all_variables()

2. **Directorio de Funciones (FunctionDirectory)**

Estructura usada: dict[str, FunctionEntry]

El uso de un diccionario permite registrar y consultar funciones rápidamente por nombre, que es su identificador único en el lenguaje. Esto facilita validar que una función no sea declarada más de una vez y acceder a sus detalles (parámetros, variables locales, tipo de retorno, etc.) cuando se necesiten durante el análisis semántico o generación de código.

Operaciones principales:

- add_function(name, return_type, param_types, start_quad)
- has_function(name)
- get_function(name)
- all_functions()

3. **Cubo Semántico (semantic_cube)**

Estructura usada: dict[str][str][str]

El cubo semántico es una matriz tridimensional representada con diccionarios anidados, lo que permite modelar de manera clara y eficiente las reglas de combinación de tipos según las operaciones aplicadas. Esta estructura es ideal para evaluar expresiones en tiempo de compilación, ya que permite consultar el tipo resultante (o detectar errores) con base en los operandos y el operador con acceso directo y sin lógica compleja adicional.

Operaciones principales:

- semantic_cube[left_type][right_type][operator]

4. **Pilas**

a. Pila de Operandos

Propósito: Almacenar las direcciones de memoria de variables, constantes o temporales que participan en las expresiones.

Acciones clave:

- push al reconocer un var, CTE_INT o CTE_FLOAT.
- pop de dos direcciones al generar un cuadruplo binario.

b. Pila de Tipos

Propósito: Mantener los tipos (int, float, bool) paralelamente a la pila de operandos, para validar con el semantic cube.

Acciones clave:

- push del tipo al ingresar un operando a la pila de operandos.
- pop de dos tipos al generar un cuadruplo y determinar el tipo resultante.

c. Pila de Operadores

Propósito: Guardar los símbolos de operador (+, -, *, /, <, >, !=, =) en espera de producir el cuadruplo correspondiente, respetando precedencia y asociación.

Acciones clave:

- push al leer un operador en la gramática.
- pop justo antes (o durante) de llamar a _generate_quad, para usar el operador correcto.

5. Fila de Cuádruplos

Estructura: Cola FIFO de tuplas (operador, op_izq, op_der, resultado).

Propósito: Registrar secuencialmente cada instrucción intermedia que luego alimentará la etapa de generación de código o ejecución.

Acciones clave:

- Cuando dos operandos y un operador están listos, se genera un nuevo cuadruplo.
- Se enfila al final de la lista quadruples.
- El índice de cuadruplo (next_quad) se incrementa para referencia de saltos en otras etapas (por ejemplo, if y while).

Tecnológico de Monterrey

Diagrama de flujo: generación de cuádruplos

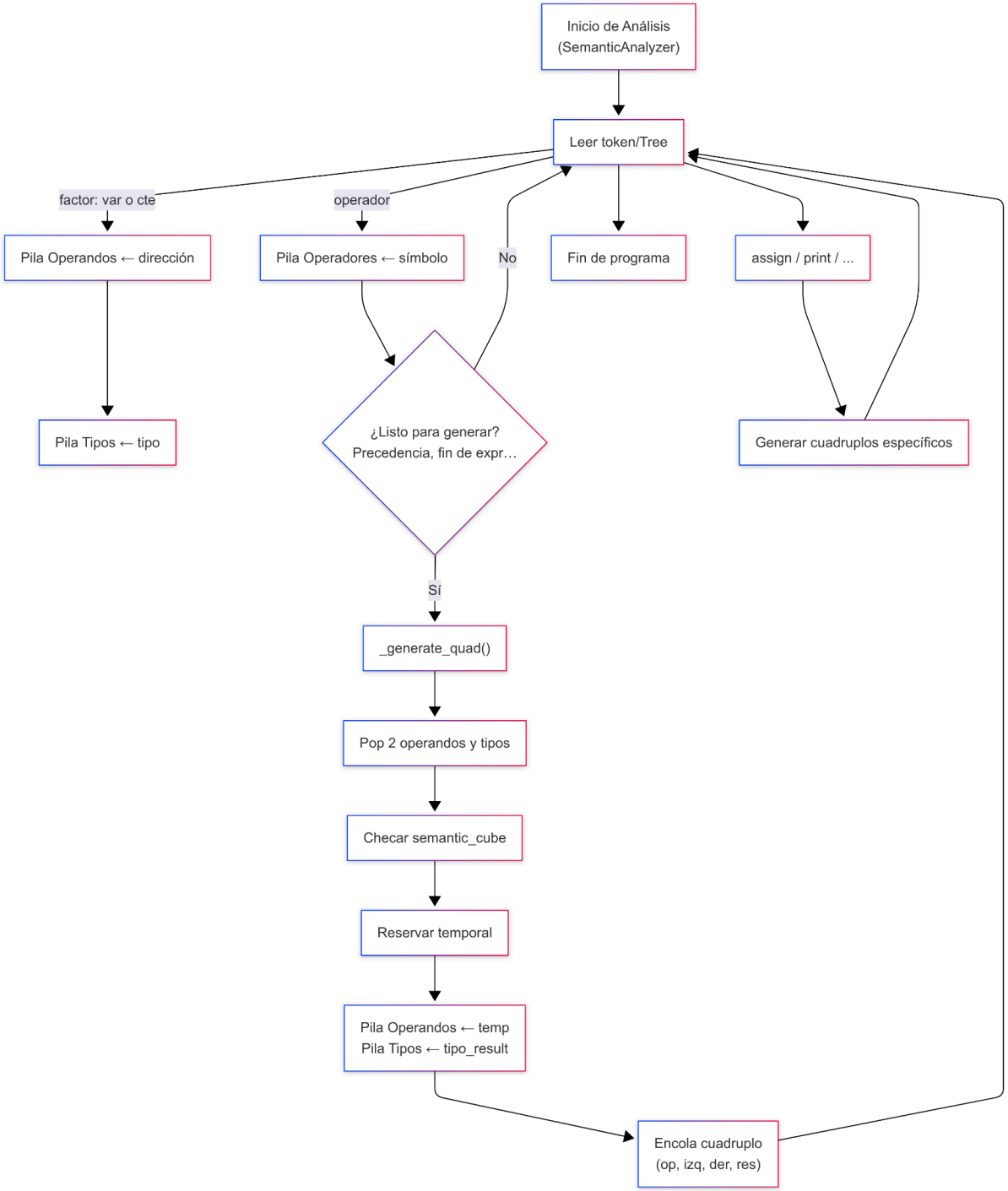
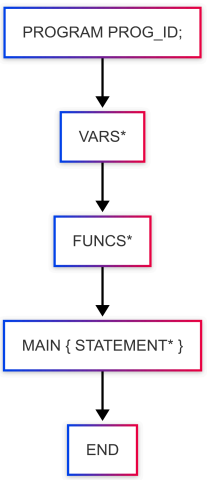


Diagrama del lenguaje BabyDuck



Puntos neurálgicos señalados en color (asumido en clase):

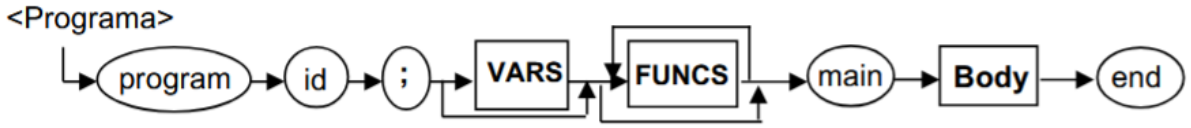
- Cabecera (PROGRAM PROG_ID;): define nombre de programa.
- Bloque VARS: declaraciones globales.
- Bloque FUNCS: cada función con su start_quad.
- MAIN: punto de entrada, inicia generación de cuádruplos para instrucciones lineales.
- END: cierre; limpia contexto.

Tecnológico de Monterrey

Descripción de acciones principales

Acción	¿Qué pasa?
Lectura de var / ID / CTE	→ Empuja en Operandos la dirección; en Tipos el tipo (int/float).
Encuentro de operador binario	→ Empuja en Operadores el símbolo; espera a que se cumpla condición de precedencia o fin de subexpresión.
Generar cuadruplo binario	1. Pop de 2 operandos y 2 tipos. 2. Valida en semantic_cube que la operación sea válida. 3. Reserva un temporal. 4. Push resultado. 5. Encola (op, izq, der, res).
Negación unaria (-factor)	→ Empuja constante 0 como operando, luego genera cuadruplo 0 - factor.
Asignación	1. Pop de resultado de expresión. 2. Resuelve dirección de la variable destino. 3. Valida compatibilidad con =. 4. Encola ('=', expr, None, var).
Print	Por cada elemento en lista: encola ('print', operando_o_literal, None, None).
Registro de función	Al ver VOID ID(...): marca start_quad = next_quad para futuros saltos. Registra parámetros en tabla local.
Fin de función / scope	Al salir del bloque body, reinicia current_function = None.

Puntos neurálgicos del lenguaje



#	Punto neurálgico	Motivo clave
1	**Decisión después de ";"** (VARS o FUNCS)	El parser debe distinguir si vienen variables globales o si brinca directo a las funciones. Un error desfasará todo lo que siga.
2	**Bucle de FUNCS**	Mientras la próxima cabecera sea de función, el analizador repite la caja. Reconocer a tiempo la palabra reservada **`main`** para salir es crítico.
3	**"end" final**	Marca el fin inequívoco del programa. Si falta o hay tokens extra después, el parser pierde sincronía y genera errores en cascada.