UNIVERSIDAD CATÓLICA DEL NORTE
ESCUELA DE INGENIERÍA
CARRERA DE ITI

CREACIÓN DE COMPILADOR MEDIANTE PLY

Profesor:
Jose Luis Veas Muñoz
Estudiantes:
Denzel Martin Delgado Urquieta
Darwin Mauricio Tapia Urrutia

06 – 2025

1. Índice

2. Introducción

Un compilador es una herramienta digital hecha para capturar código de alto nivel y traducirlo a bajo nivel para el entendimiento de una computadora. Para ello, su procesamiento pasa por tres fases generalmente, análisis, optimización y generación de código. El análisis del código fuente es verificado a través de dos componentes principales, el analizador léxico y analizador sintáctico; la optimización de código hace referencia a una ejecución más rápida y eficiente respecto a las mejoras que se le hagan al código fuente; por último, la generación de código, que consiste en la producción de un archivo ejecutable en el lenguaje de la máquina destino.

Por otro lado, un lenguaje de programación se puede entender como un conjunto de instrucciones que están regidas por las reglas establecidas de un compilador, es decir, el analizador léxico y sintáctico del mismo.

Por lo tanto, ambos conceptos están totalmente entrelazados si se requiere que la máquina realice una tarea específica solicitada por el programador. Dicho de otro modo, el desarrollador escribe el código fuente con un lenguaje de programación determinado, y el compilador comprueba las instrucciones para posteriormente ejecutarlas como tareas para un computador.

Entonces, en esta oportunidad, se desarrolló un compilador mediante PLY que permite un lenguaje de programación personalizado, con la intención de explicar detalladamente la creación de este. Se profundizará el analizador léxico, analizador sintáctico junto con el árbol AST, generación de código, lenguaje de programación creado y un manual de usuario del lenguaje inventado.

3. Objetivos

Objetivo general:

- ✔ Evaluar y Reconocer la implementación del compilador y sus funcionalidades particulares

Objetivos específicos:

- ✔ Dar a conocer las partes que componen la elaboración de un lenguaje de programación según el formato dado.
- ✔ Comprender el analizador léxico y sintáctico de PLY.
- ✔ Explicar la sintaxis personalizada del lenguaje creado.
- ✔ Describir el proceso de generación de código.
- ✔ Analizar y Exponer la construcción del árbol AST

4. Gramática del Lenguaje

La gramática define la sintaxis del lenguaje, estableciendo las reglas que debe seguir el código para ser considerado válido. Se compone de un conjunto de terminales (tokens) y reglas de producción.
Los tokens son las unidades más pequeñas del lenguaje, identificadas por el analizador léxico (lexer.py). Incluyen:

- **Identificadores y Literales:**
  - IDENTIFICADOR: Nombres de variables (e.g., entero, suma).
  - NUMERO: Valores numéricos, tanto enteros como de punto flotante (e.g., 100, 25.5).
  - CADENA: Texto encerrado en comillas dobles (e.g., "Hola, ").
- **Palabras Reservadas (Keywords):**
  - **Asignación:** devote
  - **Funciones:** decree (definir función), yield (retornar valor).
  - **Operadores Aritméticos:** inherit (+), plunder (-), forge (*), cleave (/), shatter (%)
  - **Estructuras de Control:** judge (if), exile (else), vigil (while), march (for)
  - **Funciones Integradas:** print, unir (concatenar cadenas), inquire (entrada de usuario), parias (función de impuesto), conquistar (función de simulación)
  - **Operador Unario:** menos (negación)
- **Operadores Lógicos y de Comparación:**
  - > (MAYOR), < (MENOR), >= (MAYORIGUAL), <= (MENORIGUAL)
  - == (IGUAL), != (DESIGUAL)
  - && (AND), || (OR), ! (NOT)
- **Símbolos de Puntuación:**
  - ( ) (PARIZQ, PARDER)
  - { } (LLAVEIZQ, LLAVEDER)
  - ; (PUNTOYCOMA)
  - , (COMA)

Reglas de Producción (Sintaxis)

Estas reglas (yacc.py) establecen cómo se combinan los tokens para formar estructuras válidas, desde una simple asignación hasta una definición de función completa.

- **Programa (inicio):** Un programa puede contener sentencias o declaraciones de función en cualquier orden.
  inicio : | sentencia inicio
  | declaracion_funcion inicio

- **Estructuras de Control:**
  - **Condicional judge/exile**: Ejecuta bloques de código basados en una condición booleana. Se demuestra en prueba.txt para la validación de edad y valores secretos.
  - **Bucle vigil**: Repite un bloque de código mientras una condición sea verdadera, como se usa en el contador de prueba.txt.
  - **Bucle march**: Un bucle for clásico con inicializador, condición y actualizador, usado para iterar un número fijo de veces en prueba.txt.
- **Funciones:**
  - **Declaración (declaracion_funcion):** Define una función con decree, como imprimirSuma y cambiar en programa.txt.
  - **Llamada (expresion_llamada_funcion):** Invoca una función previamente definida, como la llamada a imprimirSuma(5, 3).
  - **Retorno (sentencia_yield):** Devuelve un valor desde una función, como lo hace yield resta; en la función cambiar.

5. Diseño del AST

El AST es una representación jerárquica del código fuente que captura su estructura semántica. Cada nodo del árbol corresponde a una construcción del lenguaje. El AST es generado por el analizador sintáctico (parser) y se define a través de un conjunto de clases en yacc.py.

A continuación, se describen los nodos principales:

- **BlockNode**: Representa un bloque de sentencias secuenciales. Es el nodo raíz del árbol. También se utiliza para agrupar sentencias dentro de estructuras de control como if , else , while y for.
- **FunctionDefNode**: Representa la definición de una función con decree. Almacena el nombre de la función, una lista con los nombres de sus parámetros y un BlockNode que contiene el cuerpo de la función.
- **FunctionCallNode**: Representa la invocación de una función. Guarda el nombre de la función a la que se llama y una lista de nodos de expresión que son sus argumentos.
- **ReturnNode**: Corresponde a la sentencia yield. Contiene un nodo de expresión opcional para el valor que se devolverá.
- **MultiPrintNode**: Una versión mejorada del nodo PrintNode. Ahora contiene una lista de expresiones, permitiendo que la función print maneje múltiples argumentos.
- **LiteralNode**: Representa un valor constante, como un número (100) o una cadena de texto ('Hola, ').
- **IdentifierNode**: Representa el uso de una variable o identificador (e.g., entero).
- **AssignmentNode**: Corresponde a una operación de asignación con devote. Contiene un IdentifierNode para la variable y un nodo de expresión para el valor a asignar.
- **BinaryOpNode**: Representa una operación binaria, como la suma (inherit), resta (plunder), multiplicación (forge), división (cleave), o una operación lógica como &&.
- **UnaryOpNode**: Representa una operación unaria, como la negación con menos (UMINUS en el AST).
- **PrintNode**: Representa una llamada a la función print, conteniendo la expresión a imprimir.
- **IfNode**: Corresponde a la estructura judge/exile. Contiene un nodo para la condición, un BlockNode para el cuerpo if y opcionalmente otro para el cuerpo else.

- **WhileNode**: Representa el bucle vigil. Contiene un nodo de condición y un BlockNode para el cuerpo del bucle.
- **ForNode**: Representa el bucle march. Almacena los nodos para la inicialización, la condición, la actualización y el BlockNode del cuerpo.
- **InputNode**: Representa la función inquire para leer la entrada del usuario.
- **PariasCallNode** y **ConquistarCallNode**: Nodos específicos para las funciones personalizadas parias y conquistar, respectivamente.

6. Proceso de generación de código

El lenguaje es interpretado mediante el recorrido del AST. El mecanismo de ejecución está diseñado para soportar tanto scripts simples como funciones con alcance de variables (scope) a través de una **pila de contextos (context_stack)**.

I. Fases de Análisis:

El código se procesa con un analizador léxico y sintáctico para generar el AST. Los resultados intermedios se guardan en lexer_output.txt y ast_output.txt.

II. Fase de Ejecución (Interpretación del AST):

A. **Pila de Contextos:** La ejecución se gestiona con una context_stack, que es una lista de diccionarios. El primer elemento es siempre el **contexto global**.
B. **Ejecución simple:** Para código sin funciones de usuario, todas las operaciones ocurren en el contexto global. Las variables se leen y escriben en el único diccionario de la pila.
C. **Ejecución con Funciones:**
    1. **Definición (decree):** Al definir una función, su FunctionDefNode se almacena en el contexto global, sin ejecutar su cuerpo.
    2. **Llamada a Función:**
        a) Se crea un **nuevo contexto** (diccionario) para el alcance local de la función.
        b) Los valores de los argumentos pasados en la llamada se asignan a los parámetros en este nuevo contexto.
        c) Este nuevo contexto se **empuja a la cima de la pila** (context_stack.append()).
        d) El cuerpo de la función se ejecuta. Ahora, cualquier variable se buscará primero en este contexto local antes de pasar a los contextos inferiores (como el global).

3. **Retorno (yield):** Cuando se ejecuta yield, se lanza una excepción ReturnValue con el valor a devolver.
4. **Finalización de la Llamada:** La llamada a la función captura la excepción, obtiene el valor y, crucialmente, **saca el contexto local de la pila** (context_stack.pop()), limpiando el alcance de la función y evitando que sus variables locales "se filtren".

7. Ejemplos de código

Código fuente:

```
decree imprimirSuma(num1, num2) {
    res devote num1 inherit num2;
    print("La suma es: ", res);
}

imprimirSuma(5, 3);

decree cambiar(entero) {
    resta devote entero plunder 10;
    yield resta;
}

num devote 20;
print(num);
num devote cambiar(num);
print(num);
```

Salida del Lexer (lexer_output.txt):

TIPO	VALOR	LINEA	POSICION
DECREE	decree	1	0
IDENTIFICADOR	imprimirSuma	1	7
PARIZQ	(	1	19
IDENTIFICADOR	num1	1	20
COMA	,	1	24
PRINT	print	16	257
PARIZQ	(	16	262
IDENTIFICADOR	num	16	263
…	…	…	…
PUNTOYCOMA	;	16	267

Estructura AST:

```
BlockNode
|   FunctionDefNode: decree imprimirSuma(num1, num2)
|       BlockNode
|       |   AssignmentNode: devote
|       |   |   IdentifierNode: res
|       |       BinaryOpNode: inherit
|       |       |   IdentifierNode: num1
|       |           IdentifierNode: num2
|           MultiPrintNode
|           |   LiteralNode: 'La suma es: '
|               IdentifierNode: res
|   FunctionCallNode: imprimirSuma
|   |   LiteralNode: 5
|       LiteralNode: 3
|   FunctionDefNode: decree cambiar(entero)
|       BlockNode
|       |   AssignmentNode: devote
|       |   |   IdentifierNode: resta
|       |       BinaryOpNode: plunder
|       |       |   IdentifierNode: entero
|       |           LiteralNode: 10
|           ReturnNode: yield
|               IdentifierNode: resta
|   AssignmentNode: devote
|   |   IdentifierNode: num
|       LiteralNode: 20
|   MultiPrintNode
|       IdentifierNode: num
|   AssignmentNode: devote
|   |   IdentifierNode: num
|       FunctionCallNode: cambiar
|           IdentifierNode: num
    MultiPrintNode
        IdentifierNode: num
```

Traza de Ejecución Detallada:

1. **Contexto Inicial:** context_stack = [{'__builtins__': ...}] (Contexto global).
2. Se evalúa decree cambiar(entero) {...}. El objeto FunctionDefNode de cambiar se almacena en el contexto global. context_stack queda como [{..., 'cambiar': <FunctionDefNode>}].
3. Se evalúa num devote 20;. La variable num se asigna en el contexto global. context_stack es [{..., 'cambiar': <...>, 'num': 20}].
4. Se evalúa print(num);. Se busca num (se encuentra 20 en el global) y se imprime 20.
5. Se evalúa la asignación num devote cambiar(num);. Se debe resolver primero el lado derecho:
    - **Llamada a cambiar(num)**: a. Se busca num en el contexto global (valor 20). b. Se crea un **nuevo contexto local**: {}. c. El valor 20 se asigna al parámetro entero en el nuevo contexto: {'entero': 20}. d. El nuevo contexto se apila: context_stack ahora es [<global>, {'entero': 20}]. e. **Se ejecuta el cuerpo de cambiar**: i. resta devote entero plunder 10;. Se busca entero (se encuentra 20 en el contexto local), se calcula 20 - 10 = 10, y se asigna a resta en el contexto local. El contexto local es ahora {'entero': 20, 'resta': 10}. ii. yield resta;. Se busca resta (se encuentra 10) y se lanza ReturnValue(10).
    - **Retorno de la llamada**: a. La FunctionCallNode captura la excepción y extrae el valor 10. b. El contexto local de cambiar se saca de la pila (pop). context_stack vuelve a ser [<global>]. c. La llamada a cambiar(num) devuelve como resultado el valor 10.
6. La asignación finaliza: num devote 10;. El num en el contexto global se actualiza a 10.
7. Se evalúa el último print(num);. Se busca num (se encuentra 10) y se imprime 10.

8. Bibliografía

Beazley, D. (2020, 22 abril). *PLY (Python Lex-Yacc) — ply 4.0 documentation*.

Recuperado 9 de junio de 2025, de https://ply.readthedocs.io/en/latest/