TECNOLÓGICO
NACIONAL DE MÉXICO

Propuesta Proyecto Pseudo - Compilador en Python

Materia: Lenguajes y Autómatas II
Maestro: Ricardo González González
Grupo: A Semestre: 8

Integrantes:

- Chaires Pasalagua Luis Farid
- Oscar Hurtado González
- Jiménez Téllez José Alfredo
- Leal Rusiles Luis Rubén

Tabla de Contenido

Requerimientos

El proyecto deberá ser capaz de compilar un archivo de texto escrito en un lenguaje de programación con los siguientes elementos.

- 2 estructuras de iteración
- 1 estructura de decisión
- 3 tipos de datos: numérico, cadena y booleano
- Identificadores
- Sentencias de lectura (entrada de datos por teclado) y escritura (mensajes por pantalla)
- Expresiones lógicas aritméticas
- Expresiones relacionales

Introducción

La presente propuesta de proyecto expone la intención de creación de un pseudo compilador usando el lenguaje de programación de Python. Dicho compilador abarcará la realización de las fases de análisis léxico, análisis sintáctico y análisis semántico.

Además, para la implementación se crearán las estructuras de datos y programación de lógica necesaria sin el uso de librerías que cumplan este propósito, esto con el fin de desarrollar un proyecto de autoría propia y de aportar a los integrantes las habilidades de programación necesarias para comprender de una manera más práctica la implementación de un compilador.

El objetivo principal es crear una herramienta sencilla, pero completa, que permita analizar código fuente de un lenguaje propio (pequeño lenguaje definido por nosotros) y verificar su corrección gramatical y semántica.

Gramática seleccionada

Dentro de la compilación de un lenguaje de programación, existen diversos tipos de gramáticas que se pueden definir para el análisis de este. Estas gramáticas forman parte de una clasificación conocida como la **jerarquía de Chomsky**, propuesta por el lingüista y científico computacional **Noam Chomsky** en la década de 1950. Chomsky desarrolló esta jerarquía como un marco teórico para clasificar lenguajes formales según su complejidad y las reglas que los generan, sentando las bases para la teoría de autómatas

y la informática teórica moderna. A continuación, se describen los tipos de gramáticas según esta clasificación:

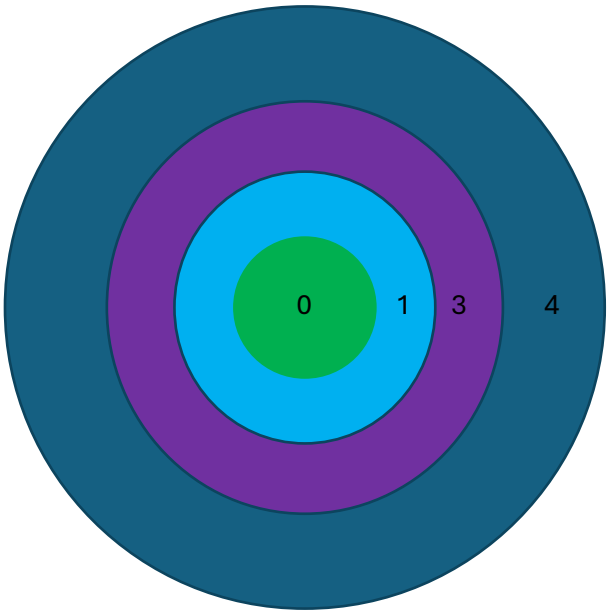
1. **Tipo 0 (Irrestrictas):**
    a. **Definición:** Gramáticas con reglas sin restricciones, donde cualquier cadena de símbolos puede ser reemplazada por otra sin limitaciones en su forma (e.g., α → β, donde α y β son cadenas arbitrarias).
    b. **Equivalencia:** Son equivalentes a las **Máquinas de Turing**, dispositivos teóricos capaces de resolver cualquier problema computable dado suficiente tiempo y memoria.
    c. **Ejemplo:** Una regla como aBc → de es válida, sin importar el contexto de los símbolos.
    d. **Uso:** Aunque son muy poderosas, su complejidad las hace poco prácticas para compiladores reales.
2. **Tipo 1 (Sensibles al contexto):**
    a. **Definición:** Las reglas pueden depender del contexto en que aparezcan los símbolos, es decir, una producción tiene la forma αAβ → αγβ, donde A es un no terminal y el contexto (α y β) restringe la aplicación de la regla.
    b. **Equivalencia:** Corresponden a las **máquinas de Turing con cinta acotada** o autómatas lineales acotados.
    c. **Ejemplo:** aBc → adef solo se aplica si a y c están presentes como contexto.
    d. **Uso:** Útiles para lenguajes naturales, pero demasiado complejas para la mayoría de los lenguajes de programación.
3. **Tipo 2 (Libres de contexto):**
    a. **Definición:** Las reglas tienen la forma A → α, donde A es un no terminal y α es una combinación arbitraria de terminales y no terminales. No dependen del contexto circundante.
    b. **Equivalencia:** Equivalentes a los **autómatas de pila**, estructuras que usan una pila para manejar recursión y anidamiento.
    c. **Ejemplo:** S → aSb | ε genera el lenguaje {a^n b^n | n ≥ 0}, como ab, aabb, etc.
    d. **Uso:** Es el tipo más utilizado en compiladores debido a su balance entre expresividad y facilidad de análisis.
4. **Tipo 3 (Regulares):**
    a. **Definición:** Las reglas son muy restringidas, de la forma A → aB o A → a, generando lenguajes regulares que pueden describirse con expresiones regulares.
    b. **Equivalencia:** Corresponden a los **autómatas finitos**, ideales para tareas simples como el análisis léxico.
    c. **Ejemplo:** S → aS | b genera el lenguaje de todas las cadenas que terminan en b (e.g., ab, aab).
    d. **Uso:** Perfectas para tokenización, pero insuficientes para estructuras sintácticas complejas.

Clasificación y diagrama de matrioska

La jerarquía de Chomsky se visualiza como un conjunto de "muñecas rusas" o matrioskas, donde cada nivel está contenido dentro del anterior en términos de poder expresivo:

- **Tipo 0** (Irrestrictas) contiene a todas las demás, ya que no tiene límites.
- **Tipo 1** (Sensibles al contexto) está dentro del Tipo 0, pero restringe las reglas al contexto.
- **Tipo 2** (Libres de contexto) está dentro del Tipo 1, limitándose a reglas sin contexto.
- **Tipo 3** (Regulares) es el más restringido, contenido dentro del Tipo 2.

Textualmente, esto se representa como: **Tipo 3 ⊆ Tipo 2 ⊆ Tipo 1 ⊆ Tipo 0**, donde cada subconjunto es menos poderoso, pero más fácil de procesar computacionalmente.



Selección de una GLC de tipo 2

Para la definición de la gramática del lenguaje en este proyecto, se optó por una **gramática libre de contexto (GLC) de tipo 2**. Esta decisión se fundamenta en razones teóricas y prácticas:

- **Matemáticamente:** Las GLC son reconocidas por autómatas de pila, que permiten manejar estructuras anidadas como expresiones matemáticas ((a + b) * c) o bloques de código (if { ... }), comunes en lenguajes de programación. Por ejemplo, el lenguaje {a^n b^n | n ≥ 0} no puede ser generado por una gramática regular (Tipo 3), pero sí por una GLC como S → aSb | ε. Esto demuestra su capacidad para modelar recursión, esencial en sintaxis.
- **Prácticamente:** Los algoritmos de análisis sintáctico como **LL(k)** y **LR(k)**, ampliamente usados en compiladores, operan eficientemente con GLC. Por ejemplo, una regla como Exp → Exp + Term | Term permite analizar expresiones aritméticas de manera determinista.

La realización de la actividad sobre análisis sintáctico asignada en el parcial 2 nos permitió familiarizarnos con el proceso requerido para la definición de una gramática mediante la implementación de estructuras de datos como árboles (para representar la estructura sintáctica) y pilas (para el análisis descendente o ascendente). Esta experiencia práctica reforzó la elección de una GLC como base del proyecto.

Elementos de una gramática libre de contexto (GLC) de tipo 2

Una gramática libre de contexto se define formalmente como una cuádrupla G = (V, T, P, S), donde cada elemento se describe como sigue:

1. **V: Conjunto de variables (o no terminales)**
    a. **Definición:** Es un conjunto finito de símbolos que representan categorías sintácticas o construcciones abstractas del lenguaje. Estos no terminales se utilizan para generar cadenas mediante reglas de producción y eventualmente se reemplazan por terminales o combinaciones de terminales y no terminales.
    b. **Propósito:** Actúan como placeholders o nodos intermedios en la derivación de una cadena válida.
    c. **Ejemplo:** En una gramática para expresiones aritméticas, V = {E, T, F}, donde E representa una expresión, T un término y F un factor.
    d. **Notación común:** Se denotan con letras mayúsculas (por ejemplo, S, A, B).
2. **T: Conjunto de terminales**
    a. **Definición:** Es un conjunto finito de símbolos que constituyen las "palabras" o elementos básicos del lenguaje. Estos son los símbolos que aparecen en la cadena final generada y no se reemplazan ulteriormente.
    b. **Propósito:** Representan los componentes concretos del lenguaje, como palabras reservadas, operadores o caracteres en un programa.
    c. **Ejemplo:** Para expresiones aritméticas, T = {+, *, (,), a, b, c}, donde a, b, c son identificadores y +, *, (,) son operadores y delimitadores.
    d. **Notación común:** Se denotan con letras minúsculas o símbolos (por ejemplo, a, +).

3. **P: Conjunto de producciones (o reglas de producción)**
    a. **Definición:** Es un conjunto finito de reglas que especifican cómo se pueden reemplazar los no terminales. Cada regla tiene la forma $A \rightarrow \alpha$, donde A es un no terminal ($A \in V$) y α es una cadena de terminales y/o no terminales, incluyendo la cadena vacía ε.
    b. **Propósito:** Define las transformaciones permitidas para generar cadenas válidas del lenguaje a partir del símbolo inicial.
    c. **Ejemplo:** Para expresiones aritméticas:
        i. $E \rightarrow E + T$
        ii. $E \rightarrow T$
        iii. $T \rightarrow T * F$
        iv. $T \rightarrow F$
        v. $F \rightarrow (E)$
        vi. $F \rightarrow a$
        vii. $F \rightarrow b$
        viii. $F \rightarrow c$

        Estas reglas permiten derivar expresiones como a + b * c.
    d. **Característica clave:** En una GLC, el lado izquierdo de cada producción debe ser un solo no terminal, lo que la distingue de las gramáticas sensibles al contexto.
4. **S: Símbolo inicial (o símbolo de arranque)**
    a. **Definición:** Es un no terminal especial ($S \in V$) desde el cual comienza toda derivación del lenguaje.
    b. **Propósito:** Sirve como punto de partida para generar todas las cadenas válidas del lenguaje definido por la gramática.
    c. **Ejemplo:** En el ejemplo anterior, S = E, ya que todas las expresiones aritméticas válidas se derivan de E.
    d. **Notación común:** Suele denotarse como S, aunque puede variar según el contexto.

Propiedades y características adicionales

- **Libre de contexto:** Las reglas de producción no dependen de los símbolos circundantes al no terminal que se está reemplazando, lo que simplifica el análisis sintáctico y permite el uso de autómatas de pila para reconocer el lenguaje.
- **Lenguaje generado:** El lenguaje L(G) generado por una GLC G consiste en todas las cadenas de terminales que se pueden derivar a partir de S aplicando las reglas de P en cualquier orden, hasta que no queden no terminales.

Ejemplo completo

Consideremos una GLC simple para el lenguaje {a^n b^n | n ≥ 0} (cadenas con igual número de a y b, como ε, ab, aabb, etc.):

- V = {S}
- T = {a, b}
- P = {S → aSb, S → ε}
- S = S

Derivación de aabb:
S → aSb → aaSbb → aabb (reemplazando S → ε en el último paso).

Este ejemplo ilustra cómo los elementos trabajan juntos: S inicia la derivación, las reglas en P generan la estructura anidada, y T proporciona los símbolos finales.

Marco teórico conceptual

Contexto

En el ámbito de las gramáticas formales, el "contexto" refiere a los símbolos circundantes que condicionan la aplicación de una regla. En las gramáticas sensibles al contexto (Tipo 1), el contexto es explícito (e.g., aA → ab solo se aplica si a precede a A). En las GLC (Tipo 2), las reglas son independientes del contexto, lo que simplifica el análisis al eliminar dependencias externas, pero limita su poder expresivo frente a las Tipo 1.

Máquina de Turing

Propuesta por **Alan Turing** en 1936, la Máquina de Turing es un modelo teórico de computación que consta de una cinta infinita, un cabezal de lectura/escritura y un conjunto de reglas. Es capaz de simular cualquier algoritmo y, por ende, reconocer lenguajes irrestrictos (Tipo 0). Su relevancia radica en que establece el límite superior de lo computable, sirviendo como base para entender la jerarquía de Chomsky.

Ejemplos de gramáticas

- **Tipo 0:** aBc → de genera lenguajes no restringidos, como {a^n b^n c^m | n, m ≥ 0} con transformaciones arbitrarias.
- **Tipo 1:** aBc → adef genera {a^n de f^n | n ≥ 1}, dependiendo del contexto a y f.
- **Tipo 2:** S → aSb | ε genera {a^n b^n | n ≥ 0}, ideal para estructuras balanceadas.
- **Tipo 3:** S → aS | b genera {a^* b}, útil para patrones lineales simples.

Python para la programación

Para la programación del pseudo compilador nos decidimos por Python, esto por diversas razones que expondremos a continuación.

Un motivo fuerte de la elección es el hecho de la facilidad para escribir y manipular ciertas declaraciones que tienen que ver con la iteración sobre arreglos o listas, además, Python cuenta con una sintaxis más directa y concisa para la realización de actividades como la iteración sobre colecciones de datos y creación de estructuras de datos.

La facilidad de escritura de código en los puntos mencionados anteriormente, lo complementa el hecho que es un lenguaje que puede orientarse a una programación a objetos, por tanto, nos permitirá alinear las ideas que tenemos pensadas para la implementación del proyecto a través de un paradigma con el que todos los integrantes del equipo se encuentran familiarizados.

Componentes del lenguaje

Reglas de Nominación

- **Inicio:** Identificadores comienzan con @ (e.g., @x).
- **Fin:** No terminan en operadores ni delimitadores (e.g., @x+ es inválido).
- **Restricciones:** No usar palabras reservadas (e.g., @While es inválido).
- **Alfabeto permitido:** Solo letras y números del alfabeto definido, sin ñ.
- **Delimitador de sentencia:** Punto (.).
- **Ejemplos válidos:** @edad, @suma12, @flag.
- **Ejemplos inválidos:** edad (sin @), @If (reservada), @x+ (termina en operador), @variableMuyLarga123 (>15).

Alfabeto

- { "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "+" | "-" | "*" | "/" | "=" "<" | ">" "&" | "|" | "!" | "(" | ")" | "{" | "}" | "." | " " | "@" | "\t" | "\n" }

Total: 74 símbolos, incluyendo letras, números, operadores, delimitadores y espacios.

Tokens

- **Caracter ::=** "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
- **Número ::=** "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
- **Espacio ::=** " " | "\t" | "\n"

- **Identificador ::=** @ Caracter (Caracter | Número)* .
  Ejemplo: @x., @var12.
- **Palabra Reservada ::=** Num | Text | Bool | While | For | If | Else | Read | Write | True | False
- **Tipo_dato ::=** Num | Text | Bool
- **Num ::=** Número+ (e.g., 123)
- **Text ::=** Caracter (Caracter | Número)* (e.g., abc123)
- **Comentario ::=** "$" hasta "\n" (e.g., $ Comentario)
- **Bool ::=** True | False
- **Operador aritmético ::=** "+" | "-" | "*" | "/"
- **Operador relacional ::=** "==" | "!=" | "<" | ">" | "<=" | ">="
- **Operador lógico ::=** "&&" | "||" | "!"
- **Operador de asignación ::=** "="
- **Operador de incremento ::=** "++"
- **Operador de decremento ::=** "--"
- **Delimitador ::=** "(" | ")" | "{" | "}" | "."

Reglas de la Gramática

- **Programa → ListaSentencias**
  - **Ejemplo:** Num @x. @x = 5. Write(@x). (programa simple).
- **ListaSentencias → Sentencia ListaSentencias | ε**
  - **Ejemplo:** Num @x. @x = 5. (dos sentencias) | ε (programa vacío).
- **Sentencia → Declaración | Asignación | IfSent | WhileSent | ForSent | ReadSent | WriteSent**
- **Declaración → (Num | Text | Bool) Identificador . | (Num | Text | Bool) Identificador = Expresión .**
  - **Ejemplo:** Num @x. (declaración simple) | Text @msg = hola. (con asignación).
- **Asignación → Identificador = Expresión .**
  - **Ejemplo:** @x = 5 + 3. (asigna 8 a @x).
- **IfSent → If (Condición) { ListaSentencias } | If (Condición) { ListaSentencias } Else { ListaSentencias }**
  - **Ejemplo:** If (@x > 0) { Write(@x). } Else { Write(0). } (imprime @x si es positivo, 0 si no).
- **WhileSent → While (Condición) { ListaSentencias }**
  - **Ejemplo:** While (@i < 5) { @i = @i + 1. Write(@i). } (imprime 1 a 5).
- **ForSent → For (Asignación Condición . Asignación) { ListaSentencias }**
  - **Ejemplo:** For (@i = 0. @i < 3. @i = @i + 1) { Write(@i). } (imprime 0, 1, 2).
- **ReadSent → Read (Identificador) .**
  - **Ejemplo:** Read(@x). (lee un valor para @x).
- **WriteSent → Write (Expresión) .**

    - **Ejemplo:** Write(@x + 1). (imprime @x incrementado).
- **Condición → Expresión Operador_relacional Expresión | Expresión Operador_lógico Expresión | (Condición)**
    - **Ejemplo:** @x > 5 && @y == 0 (verdadero si @x > 5 y @y = 0).
- **Expresión → Expresión + Término | Expresión - Término | Término**
    - **Ejemplo:** 5 + 3 - 2 (evalúa a 6).
- **Término → Término * Factor | Término / Factor | Factor**
    - **Ejemplo:** 4 * 2 / 2 (evalúa a 4).
- **Factor → Identificador | Num | (Expresión)**
    - **Ejemplo:** @x, 10, (5 + 3).