Tecnológico de Monterrey

Desarrollo de aplicaciones avanzadas de ciencias computacionales (503)

Patito - Entrega #1

TC3002B.503

Mario Alberto González Méndez - A00832313

Prof. Carlos Acuña Ocampo
Prof. José Carlos Ortiz Bayliss
Profa. Elda Guadalupe Quiroga González

15 de Octubre del 2025

1. Análisis de Herramientas de Generación

Para la implementación del analizador léxico (Scanner) y sintáctico (Parser), se investigaron las siguientes herramientas:

Herramienta	Lenguaje Base	Paradigma	Ventajas Clave
Flex / Bison	C / C++	Generador	El estándar de la industria. Muy rápido y potente.
ANTLR	Java	Generador (LL)	Muy poderoso, genera *targets* para múltiples lenguajes (Python, C++, Java), excelente para gramáticas complejas.
`ply` (Python Lex-Yacc)	Python	Biblioteca (LALR)	Ligero, se integra nativamente con Python, sintaxis muy similar a Lex/Yacc, buena documentación.

2. Selección de Herramienta: ply

Se seleccionó la biblioteca ply (Python Lex-Yacc). La justificación principal es que conecta directamente con el lenguaje de desarrollo (Python), que será utilizado para el resto del compilador.

Ventajas de ply:

Integración Nativa: Al ser una biblioteca de Python, no requiere pasos de compilación intermedios ni generación de archivos en otros lenguajes.

Sintaxis Clásica: Sigue de cerca la sintaxis y los conceptos probados de Lex/Yacc, lo cual es académicamente valioso.

Buena Documentación: El proyecto ply está bien documentado, facilitando la implementación de las reglas.

3. Implementación de Reglas Léxicas (Scanner con ply.lex)
Las expresiones regulares de la Etapa 0 se implementaron en el archivo lexer.py usando ply.lex. El formato de alta de reglas fue el siguiente:

- **Palabras Reservadas**: Se definieron en un diccionario de Python (reserved), lo que permite a ply manejarlas eficientemente.

```
reserved = {
   'programa': 'PROGRAMA',
   'si': 'SI',
   'vars': 'VARS',
   # ...etc
}
```

- **Tokens Simples**: Los operadores y delimitadores se definieron como variables t_TOKEN con una expresión regular simple.

```
t_MAS = r'\+'
t_PTOCOMA = r';'
t_LPAREN = r'\('
```

- **Tokens Complejos:** Los tokens que requieren lógica (como ID, CTE_FLOT, CTE_ENT) se definieron como funciones t_TOKEN(t):. Esto nos permite convertir el valor (ej. de string a int) y manejar las palabras reservadas.

```
def t_ID(t):
   r'[A-Za-z_][A-Za-z_0-9]*'
   # Revisa si es una palabra reservada
   t.type = reserved.get(t.value, 'ID')
   return t
```

- **Tokens Ignorados**: Se usaron las variables t_ignore y t_ignore_COMMENT para descartar espacios, tabs y comentarios.

```
t_ignore = ' \t'
t_ignore_COMMENT = r'//.*'
```

4. Implementación de Reglas Gramaticales (Parser con ply.yacc)

La Gramática Libre de Contexto (CFG) de la Etapa 0 se implementó en parser.py usando ply.yacc.

- Precedencia: La precedencia de operadores se definió explícitamente en una tupla precedence para resolver ambigüedades aritméticas y unarias.

```
precedence = (
   ('left', 'MAYOR', 'MENOR', 'DIF', ...),
   ('left', 'MAS', 'MENOS'),
   ('left', 'POR', 'DIV'),
   ('right', 'UMAS', 'UMENOS') # Tokens ficticios para unarios
)
```

- Reglas Gramaticales: Cada regla de la CFG se tradujo a una función de Python p_regla(p): con la definición de la regla en su docstring.

```
# <Programa>
def p_programa(p):
    'programa : PROGRAMA ID PTOCOMA vars_opcional ... FIN'
    pass # Lógica futura aquí

# <Exp>
def p_exp(p):
    '''exp : termino
           | exp MAS termino
           | exp MENOS termino'''
    Pass
```

- Manejo de Errores: Se implementó una función p_error(p): básica para reportar errores sintácticos al usuario, indicando el token y la línea del error.

5. Plan de Pruebas y Casos de Prueba (Test-Plan)

Para validar el funcionamiento del Scanner y Parser, se diseñó el siguiente plan de pruebas.

Test-Plan:

1. Pruebas de Léxico:
    - T-LEX-01 (Correctos): Validar que todos los tokens (keywords, operadores, IDs, constantes) son reconocidos correctamente.
    - T-LEX-02 (Comentarios): Validar que los comentarios // son ignorados.
    - T-LEX-03 (Errores): Validar que caracteres ilegales (ej. @, #, $) disparan un t_error.
2. Pruebas de Sintaxis (Camino Feliz):
    - T-SYN-01 (Estructura Mínima): Validar un programa vacío (solo programa id; inicio {} fin).
    - T-SYN-02 (Estructura Completa): Validar un programa que use VARS, FUNCS, INICIO y estatutos.
    - T-SYN-03 (Precedencia): Validar que x = 2 + 3 * 4; se interpreta correctamente.

3. Pruebas de Sintaxis (Errores):

    - T-ERR-01 (Declaración): Validar error si se usa el formato entero x; (incorrecto) en lugar de x : entero; (correcto).

    - T-ERR-02 (Punto y Coma): Validar error si se omite un PTOCOMA (;) al final de un estatuto.

    - T-ERR-03 (Estructura): Validar error si falta inicio o fin.

    - T-ERR-04 (Paréntesis): Validar error si un si(...) no tiene paréntesis de cierre.

Casos de Prueba (Test-Cases) Desarrollados:

- Test-Case: prueba.pat (Cubre T-SYN-02 y T-SYN-03)

    - Este fue el archivo principal que usamos para depurar. Valida la estructura general, declaración VARS en el formato correcto, múltiples estatutos (asigna, escribe, si/sino), expresiones aritméticas y el uso de comentarios.

```
programa mi_programa;
vars
   x, y : entero;
   z : flotante;
inicio
{
   // Prueba de asignación y precedencia
   x = 10;
   z = x + (y * 3.14);

   // Prueba de E/S con extensión de string
   escribe("El valor de z es: ", z, letrero);

   // Prueba de condición
   si (z > 10.0) {
      escribe("Es mayor");
   } sino {
      escribe("Es menor o igual");
   }
   ; // Punto y coma de la condición
}
Fin
```

- **Test-Case: error_sintaxis.pat (Cubre T-ERR-01 y T-ERR-02)**

```
programa mi_programa;
vars
   entero x;  // <-- ERROR SINTÁCTICO (debe ser 'x : entero;')

inicio
{
   x = 10  // <-- ERROR SINTÁCTICO (falta ';')
}
fin
```

Test-Case: error_lexico.pat (Cubre T-LEX-03)

```
programa mi_programa;
vars
   x : entero;
inicio
{
   // Error, @ no es un token válido
   x = 10 @ 5;
}
fin
```

- **Test-Case: error_lexico.pat (Cubre T-LEX-03)**

```
programa mi_programa;
vars
   x : entero;
inicio
{
   // Error, @ no es un token válido
   x = 10 @ 5;
}
fin
```

Link del github con todos los archivos del compilador:
https://github.com/elshavo/Compiladores-ML/tree/main/Compiladores/LenguajePatitoV1