Centro universitario de ciencias exactas e ingenierías

Alumno: Jose Ramon Hernandez Gaytan

Profesor: MICHEL EMANUEL LOPEZ FRANCO

Avances en la Construcción de tu Traductor

`

Objetivo:

Desarrollar un componente específico de un traductor (puede ser un compilador o un intérprete) para un lenguaje de programación simplificado. Este proyecto se realizará en varias fases, cada una centrada en una parte diferente del proceso de traducción.
Fase Actual: Análisis Léxico y sintactico

Descripción:
En esta fase, deberás implementar el analizador léxico para el lenguaje de programación asignado. El lenguaje tendrá una sintaxis y un conjunto de tokens definidos previamente por el instructor. El analizador léxico deberá ser capaz de leer el código fuente y convertirlo en una secuencia de tokens que serán utilizados en fases posteriores del proceso de traducción.
Requerimientos:: Basado en la especificación del lenguaje proporcionada, define los tokens que formarán parte del lenguaje. Esto incluye palabras reservadas, identificadores, literales numéricos, operadores, etc.
: Escribe tu propio analizador en el lenguaje de programación de tu elección. El analizador debe poder leer un archivo de entrada con código fuente y producir una lista de tokens.
: Tu analizador léxico debe ser capaz de manejar y reportar errores léxicos de manera adecuada, como caracteres inválidos o formatos incorrectos de tokens.
Entregables:: El código fuente de tu analizador léxico y sintactico, incluyendo cualquier estructura de datos utilizada para almacenar los tokens.
: Un breve documento que explique tu diseño, las decisiones importantes que tomaste durante la implementación, y cómo se manejan los errores léxicos y sintacticos.
: Archivos de entrada de muestra junto con la salida producida por tu analizador léxico y sintactico. Incluye casos que demuestren el manejo correcto de los tokens y también ejemplos que muestren cómo se manejan los errores léxicos.
Evaluación:
Tu tarea será evaluada en base a los siguientes criterios:

- : ¿Tu analizador léxico identifica correctamente todos los tokens definidos en la especificación del lenguaje?
- : ¿Tu analizador proporciona mensajes de error útiles y precisos para entradas inválidas?
- : ¿Tu código está bien organizado, comentado y sigue las buenas prácticas de programación? ¿La documentación proporciona una buena visión general de tu implementación y decisiones

video, pdf, presentación o web,..

1. ¿El analizador léxico identifica correctamente todos los tokens definidos?

Sí, el analizador léxico identifica todos los tokens definidos en la especificación del lenguaje:

- **Palabras reservadas**: program, var, int, float, string, if, else, print.
- **Identificadores**: [a-zA-Z_][a-zA-Z0-9_]* (validados antes que las palabras reservadas).
- **Literales**:
    - Números (NUMBER): enteros (42) y decimales (3.14).
    - Strings (STRING): entre comillas dobles ("Hola"), con detección de no cerrados.
- **Operadores**: +, -, *, /, =, ==.
- **Símbolos**: ;, :, {, }, (,).
- **Comentarios**: //....
- **Espacios**: ignorados.

Ejemplo de cobertura:

javascript

```
program Ejemplo {
  var x: int = 5 + (3.14 * 2); // ✅ Todos los tokens reconocidos
  print("Hola"); // ✅ String y función
}
```

---

2. ¿Los mensajes de error son útiles y precisos?

Sí, los mensajes de error incluyen:

- **Errores léxicos**:
    - [LÉXICO] String no cerrado en línea 3, columna 15: Indica la ubicación exacta donde se abrió un string sin cerrar.
    - [LÉXICO] Carácter inválido: '@' en línea 5, columna 9: Señala caracteres no reconocidos.
- **Errores sintácticos**:
    - [SINTÁCTICO] Se esperaba ';' pero se encontró 'var' en línea 4, columna 5: Detecta tokens faltantes o inesperados.

- [SINTÁCTICO] Tipo no válido en línea 2, columna 11: Valida tipos de datos incorrectos.

Ejemplo de error útil:

javascript

var x: strng; // Error: [LÉXICO] Tipo no válido 'strng' en línea 2, columna 8

---

3. ¿El código está bien organizado y documentado?

Sí, el código sigue buenas prácticas y está documentado:

- **Organización**:
  - **Módulos separados**: lexer.py, parser.py, main.py.
  - **Clases y funciones dedicadas**: Lexer, Parser, manejo de errores.
  - **Jerarquía lógica**: Reglas sintácticas implementadas como métodos (program(), block(), etc.).
- **Comentarios**:
  - **Docstrings**: Explican el propósito de cada función/clase.
  - **Comentarios inline**: Aclaran lógica compleja (ej: manejo de strings).
- **Buenas prácticas**:
  - Nombres descriptivos: tokenize(), expect(), parse().
  - Uso de excepciones personalizadas: LexerError, ParserError.
  - Cumple con PEP8 (formato consistente, indentación, etc.).

Ejemplo de código organizado:

```
def tokenize(self, code):
  """Genera tokens con manejo especial de strings no cerrados"""
  # Lógica clara y separación de responsabilidades
  if code[pos] == '"':
    # Manejo detallado de strings
```

Documentación y Decisiones de Diseño

La documentación incluida en el código y respuestas anteriores cubre:

1. **Decisiones clave**:

- **Orden de tokens en el lexer**: Prioridad de == sobre =.
- **Manejo manual de strings**: Para detectar no cerrados.
- **Recursividad en el parser**: Para expresiones anidadas.

2. **Ejecución**: Instrucciones claras para probar casos válidos/inválidos.
3. **Ejemplos**: Archivos de prueba con entradas y salidas esperadas.

Áreas de Mejora

1. **Errores semánticos**: No se manejan (ej: usar un string en una operación numérica).
2. **Recuperación de errores**: El parser se detiene al primer error (podría intentar recuperarse).
3. **Documentación externa**: Un README.md con ejemplos de uso mejoraría la accesibilidad.

---

ejemplos:

```
PS C:\Users\jose_\OneDrive\Escritorio\Traductor> python main.py samples/valid/correct.src

[Resultado] Análisis completado exitosamente
```

```
[LÉXICO] Carácter inválido: '"' en línea 7, columna 9
```

```
PS C:\Users\jose_\OneDrive\Escritorio\Traductor> python main.py samples/valid/declaraciones.src

[ÉXITO] Análisis completado correctamente
```

Traductor > samples > invalid > errores.src

```
1   program PruebaErrores {
2       var numero: int
3       // Error sintáctico: Falta el punto y coma
4       var texto: strng;  // Error léxico: 'strng' no es un tipo válido
5
6       numero = 5 * (3 + 2;  // Error sintáctico: Falta paréntesis de cierre
7       texto = "Hola mundo
8
9       print(numero);  // Error léxico: Comillas no cerradas
10      print(texto;    // Error sintáctico: Falta paréntesis de cierre
11  }
```

```
program EjemploCompleto {
    var nombre: string;
    var edad: int;
    var precio: float;

    nombre = "Ana López";
    edad = 30;
    precio = 99.95;

    print(nombre);
    print(edad * 2);
}
```

Traductor > samples > valid > correct.src

```
1  program MiPrograma {
2      var numero: int;
3      numero = 42;
4      print(numero);
5  }
```

Conclusión

El proyecto cumple con los objetivos de análisis léxico/sintáctico, ofrece mensajes de error precisos y sigue buenas prácticas de código. ¡Es una base sólida para expandir hacia un lenguaje más complejo!