Instituto Tecnológico de Costa Rica Primera Tarea Programada
Sede Regional San Carlos Compiladores e Intérpretes

Prof. Oscar Víquez Acuña. Análisis Sintáctico
Compilador Mini-Python

Introducción

Un compilador/intérprete es una herramienta de trabajo muy importante para cualquier estudiante de computación y normalmente dichas herramientas se utilizan sin conocer detalles de su implementación que podrían ser de ayuda para comprender mejor el funcionamiento de la herramienta. Se espera que después de este proyecto, el estudiante tenga con una mejor comprensión de los compiladores/intérpretes que utiliza y al mismo tiempo incremente sus habilidades de programación multiparadigma con énfasis en OO.

Cualquier detalle extra que no sea aclarado en esta documentación y que surja debido a las dudas de los grupos, será aclarado en su momento por el profesor sin necesidad de elaborar otro enunciado.

Definición

El proyecto consiste en crear un Analizador Sintáctico para el lenguaje denominado Mini-Python cuya sintaxis se detalla en el archivo adjunto. Este lenguaje es un subconjunto de lenguaje al estilo Python por lo que NO debe entenderse que el funcionamiento deba ser necesariamente el mismo, tanto en esta como en las etapas posteriores. El objetivo en esta etapa es realizar la fase de análisis sintáctico del compilador para dicho lenguaje en donde se espera concluir con las subfases de "scanning" "parsing" y AST. Para el desarrollo de esta etapa será necesario la utilización de la herramientas ANTLR4 en el lenguaje de programación C#.

A continuación se detallan las características del lenguaje Mini_Python.

Convenciones léxicas

Comentarios y caracteres ignorados
Los comentarios deben permitirse desde que inicie con "#" hasta que termine la línea. Deben además permitir comentarios multilínea con el formato de Python de """ comentario """.
Otros caracteres ignorados incluyen el cambio de línea, return-carry, y el espacio en blanco.

Tokens

Se deben identificar los tokens diferentes de la gramática para la resolución del problema y en aquellos que requieran el uso de letras o dígitos se definen las siguientes convenciones léxicas:

letter → [a-z A-Z _]
digit → [0-9]
id → letter {letter | digit}

Observe que el "_" está tratado como una letra por cuestiones propias del formato de los nombres de identificadores en muchos lenguajes de programación.

Constantes numéricas
El token Integer es un número entero positivo. Debe permitir además el uso de punto flotante.

Constantes carácter
Se debe permitir el uso de constantes carácter sea para referirse a uno solo de ellos o a una cadena.

Operadores
Se permitirá el uso de operadores relacionales, aritméticos y demás según se indica en la sintaxis

Scanner

El Scanner debe reconocer cada uno de los tokens mencionados anteriormente y devolverlos uno por uno según sea su ocurrencia en el archivo fuente, recordando siempre la posición en el archivo del siguiente token para retornarlo en la siguiente llamada.

El scanner debe retornar una clase Token que contenga el tipo de token, el lexema, la fila y la columna del token identificado en el archivo fuente. Esto a través del archivo de configuración para la herramienta ANTLR4

Parsing

Se deben implementar los parser necesarios para el análisis sintáctico mediante el algoritmo TOP-DOWN de descenso recursivo utilizado por la herramienta ANTLR4. La gramática que se adjunta con este documento se encuentra en formato BNF *y se solicita que sea convertida a EBNF* básicamente eliminando las recursiones y en su defecto agregando repeticiones. Podrá eventualmente ser modificada por el estudiante si lo desea, pero estas modificaciones no deben alterar su funcionamiento original y deben ser debidamente documentadas.

Los errores que se muestren deben ser identificados entre errores de scanner o de parser, lo que implica necesariamente un manejo separado del reporte de errores por defecto que realiza ANTLR4. Así mismo los mensajes de error deben ser impresos en la interfaz gráfica de la aplicación.

Interfaz Gráfica

Se desea implementar para el compilador de Mini Python una aplicación en escritorio donde básicamente se permita:

- Abrir archivos de texto de diversas fuentes locales o externas a la máquina. De forma local, se debe recordar el último directorio abierto. Debe ser posible abrir más de un archivo al mismo tiempo en el editor y que las compilaciones sean realizadas sobre el archivo abierto actualmente.
- Mostrar el texto cargado utilizando al menos el número de fila para una mejor identificación de los posibles errores.
- Mostrar errores de compilación de forma simple pero legible. Cada error debe permitir llevar el cursor a la línea donde se encuentra del editor al hacerle doble click.

- Tener una opción para compilar el código cargado por turno (botón de “play” de archivo actual)

Puntos de evaluación

La evaluación de todo el proyecto se basa en la detección de errores y en la generación del código correctamente. Para esta primera tarea, la evaluación consiste en realizar el proceso de escaneo del texto fuente (Scanner), la verificación de los tokens con respecto a la gramática (Parser) y la posterior creación de la estructura intermedia AST.

Los puntos que serán evaluados para este primer proyecto son los siguientes:

Interfaz	**25**
Se cumple con la lectura de archivos solicitada	10
Se cumple con el despliegue y funcionamiento básicos solicitado	10
Se cumple con la compilación y despliegue de errores solicitados	5
Se cumple con llevar el cursor al error con doble click	5
Scanner	**25**
Tokens válidos, Comentarios y caracteres especiales	10
Manejo de Tabulaciones del lenguaje	10
Detección de errores léxicos y despliegue en interfaz (Claridad y Ubicación)	5
Parser	**45**
Declaraciones (vars, métodos)	10
Statements (if, while, method calls, etc)	10
Expresiones (simples, complejas)	15
Detección de errores sintácticos y despliegue en interfaz (Claridad y Ubicación)	5
Documentación	**10**
Formato y Contenido	5
Ortografía y Gramática	5
	100

Documentación

La documentación deberá incluir las siguientes partes y no debe exceder las 7 hojas por un lado sin incluir la portada siguiendo el formato establecido por la carrera para este fin. El contenido debe dar énfasis en lo siguiente:

- Portada formal.
- Análisis del lenguaje (análisis de la gramática, del código que puedo crear o no puedo crear con ella)
- Soluciones e implementación. (cuestiones técnica de la implementación del proyecto)
- Resultados obtenidos. (cuadro **explicativo** de lo realizado y lo no realizado – énfasis en esto último)
- Conclusiones **del trabajo**.
- Bibliografía. (documentos con autor)

Puntos clave para la implementación del proyecto

El proyecto tiene elementos que deben considerarse como claves para la consecución exitosa del mismo. Entre los que se considerar se enumeran:

- Implementación del proyecto en C#, migrando los ejemplos que se han realizado en Clase en el lenguaje Java
- Aspectos importantes de interfaz gráfica que debe requerir un tiempo importante realizar.
- Manejo de tokens para dividir bloques en Python. Quiere decir que en Python y por tanto en Mini-Python, la forma de delimitar un bloque de código es por medio de tabulaciones denotadas en la gramática como INDENT y DEDENT. Pero no todo carácter TAB que se encuentre es un INDENT, por lo que debe investigarse sobre el manejo de estos para este tipo de lenguajes (https://github.com/yshavit/antlr-denter es un ejemplo interesante pero está en Java)

Aspectos Administrativos

- La tarea se desarrollará en grupos de máximo dos personas.
- El lenguaje de programación será *C#.*
- La fecha de entrega será el Lunes 9 de Setiembre de 2024 antes de las 10:00 pm.
- El medio de entrega será el TEC-DIGITAL.
- Cualquier intento de plagio, copias totales o parciales de otras personas o de Internet, serán castigados con nota de 0.

Programa de Ejemplo

```
“””
Ejemplo de prueba
para Análisis Sintáctico
PUEDE CONTENER ERRORES DE TABULACIÓN CAUSADOS POR EL FORMATO DE WORD

Este es un ejemplo solamente, que no asegura probar todos los elementos
sintácticos del lenguaje. DEBEN HACER MAS PRUEBAS
“””

#Calcular factorial
def calcularFac(num):
    num_aux = 0
    if (num < 1):
        num_aux = 1
    else:
        num_aux = num * (calcularFac(num-1))
    return num_aux

ventas = [100,200,300,400,500]

#promedio de elementos
def promedio(cualquier_arreglo):
    tam = len(cualquier_arreglo)
    resultado=0
    #ciclo para recorrer arreglo
    cont = 0
    sumatoria = 0
    while (cont <= tam-1):
        sumatoria = sumatoria + cualquier_arreglo[cont]
        cont += 1
    if tam > 0:
        resultado = sumatoria / tam
    return resultado

def calcularpromedioventas():
    prom = promedio(ventas)
    print("El promedio de las ventas es: ")
    print(prom)

def main():
    print("Calculo del Factorial: " + calcularFac(6))
    calcularpromedioventas()
```

Grammar for mini-Python:

1. Program := Program MainStatement | MainStatement
2. MainStatement := DefStatement | AssignStatment
3. Statement := DefStatement | IfStatement | ReturnStatement | PrintStatement | WhileStatement | AssignStatment | FunctionCallStatement
4. DefStatement := **def identifier (** ArgList **) :** Sequence
5. ArgList := **identifier** Arglist | ε
6. MoreArgs := **, identifier** MoreArgs | ε
7. IfStatement := **if** Expression **:** Sequence **else :** Sequence
8. WhileStatement := **while** Expression **:** Sequence
9. ForStatement := **for** Expression **in** ExpressionList **:** Sequence
10. ReturnStatement := **return** Expression **NEWLINE**
11. PrintStatement := **print** Expression **NEWLINE**
12. AssignStatement := **identifier =** Expression **NEWLINE**
13. FunctionCallStatement := PrimitiveExpression **(** ExpressionList **) NEWLINE**
14. ~~ExpressionStatement := ExpressionList **NEWLINE**~~
15. Sequence := **INDENT** MoreStatements **DEDENT**
16. MoreStatements := Statement MoreStatements | Statement
17. Expression := AdditionExpression Comparison
18. Comparison := Comparison (**<**|**>**|**<=**|**>=**|**==**) AdditionExpression | ε
19. AdditionExpression := MultiplicationExpression AdditionFactor
20. AdditionFactor := AdditionFactor (**+**|**-**) MultiplicationExpression | ε
21. MultiplicationExpression := ElementExpression MultiplicationFactor
22. MultiplicationFactor := MultiplicationFactor (*****|**/**) ElementExpression | ε
23. ElementExpression := PrimitiveExpression ElementAccess
24. ElementAccess := ElementAcess **[** Expression **]** | ε
25. ExpressionList := Expression MoreExpressions | ε
26. MoreExpressions := MoreExpressions **,** Expression | ε
27. PrimitiveExpression := (**-**|ε) **integer** | (**-**|ε) **float** | **charConst** | **String** | **identifier** (**(** ExpressionList **)** | ε) | **(** Expression **)** | ListExpression | **len (** Expression **)**
28. ListExpression := **[** ExpressionList **]**

LA <u>REGLA 14</u> QUEDA FUERA DE LA GRAMÁTICA PARA EL PARSER, PERO NO LA QUITAMOS DE ESTE DOCUMENTO POR SI MODIFICÁRAMOS EN ALGÚN MOMENTO LA MISMA DURANTE EL DESARROLLO DE ALGUNO DE LOS PROYECTOS FUTUROS DEL CURSO