INSTITUTO TECNOLOGICO DE MEXICO

CAMPUS PACHUCA

LENGUAJES Y AUTOMATAS

5.1 Requerimiento de la gramatica

Baume Lazcano Rodolfo

Gabriela Dyvheke Espinosa Antelis

Andres Garcia Cruz

28 Mayo 2024

Descripción Detallada de la Gramática del Lenguaje

1. Introducción

Este proyecto desarrolla un analizador léxico y sintáctico para un lenguaje de programación simple, utilizando Python y la biblioteca Tkinter para crear una interfaz gráfica. El analizador ayuda a los usuarios a identificar y corregir errores léxicos y sintácticos en el código fuente que escriben.

Propósito:

El analizador léxico convierte el código fuente en una secuencia de tokens, que son las unidades léxicas significativas como palabras clave, operadores y delimitadores. El analizador sintáctico luego verifica que esta secuencia de tokens cumpla con las reglas gramaticales del lenguaje, asegurando que el código esté estructurado correctamente para su posterior compilación o interpretación.

2. Símbolos Terminales

- Palabras clave (Tokens): Estas representan palabras reservadas del lenguaje de programación, como if, else, while, for, etc., que tienen un significado específico y definen la estructura del código.
- Operadores (Tokens): Representan operaciones como asignación (=), suma (+), resta (-), multiplicación (*), etc., que se utilizan para realizar operaciones sobre variables y valores.
- Símbolos de puntuación (Tokens): Estos son caracteres especiales como punto y coma (;), paréntesis ((,)), llaves ({, }), etc., que se utilizan para delimitar estructuras de código y expresiones.
- Identificadores (Tokens): Representan nombres de variables, funciones u otros elementos definidos por el usuario en el código fuente. Estos identificadores se utilizan para hacer referencia a diferentes partes del programa.
- Valores numéricos (Tokens): Representan números enteros y flotantes, que se utilizan para asignar valores a variables o como parte de expresiones matemáticas dentro del código fuente.

Estas producciones guían al analizador sintáctico para entender cómo los tokens deben estructurarse en el código fuente.

2. Símbolos No Terminales

- **parse_statement():** Esta función es responsable de analizar la estructura de las declaraciones e instrucciones en el código fuente. Dentro de esta función, se determina el tipo de declaración o instrucción presente y se llama a la función correspondiente para analizarla más a fondo.
- **parse_declaration():** Esta función analiza la estructura de las declaraciones de variables en el código fuente. Verifica si una secuencia de tokens representa una declaración válida de variable, incluyendo el tipo de dato, el nombre de la variable y, en algunos casos, el valor asignado a la variable.
- **parse_if_statement():** Esta función analiza la estructura de las instrucciones condicionales if en el código fuente. Se encarga de verificar si la sintaxis del if es correcta, incluyendo la condición y el bloque de código que se ejecuta si la condición es verdadera. Además, también analiza la estructura del bloque else, si está presente.
- **parse_while_statement():** Similar a parse_if_statement(), esta función analiza la estructura de las instrucciones de bucle while en el código fuente. Verifica si la sintaxis del while es correcta, incluyendo la condición y el bloque de código que se ejecuta mientras la condición sea verdadera.
- **parse_for_statement():** Al igual que las anteriores, esta función analiza la estructura de las instrucciones de bucle for en el código fuente. Se encarga de verificar si la sintaxis del for es correcta, incluyendo las partes de inicialización, condición y actualización del bucle, así como el bloque de código que se ejecuta en cada iteración.

Los símbolos no terminales representan estructuras sintácticas complejas que se descomponen en otros símbolos terminales y no terminales.

4. Producciones y Reglas

Las producciones son reglas que definen cómo se descomponen los símbolos no terminales en otros símbolos. Estas reglas forman la gramática del lenguaje.

- **parse_statement():** Esta función tiene la siguiente producción:
  - **Reglas:**

    Si la palabra clave es una declaración de variable (int, float, char, long), se llama a parse_declaration().

    Si la palabra clave es un if, se llama a parse_if_statement().

    Si la palabra clave es un while, se llama a parse_while_statement().

    Si la palabra clave es un for, se llama a parse_for_statement().

    Si no coincide con ninguna de las anteriores, se considera una expresión y se llama a parse_expression().

- **parse_declaration():** Esta función tiene la siguiente producción:
    - **Reglas:**

  Verifica que la palabra clave sea un tipo de dato válido (int, float, char, long).

  Verifica que después de la palabra clave de tipo de dato haya un identificador de variable.

  Verifica si hay un signo de igual (=) seguido de una expresión opcional para la inicialización de la variable.

  Verifica si hay un punto y coma (;) al final de la declaración.

- **parse_if_statement():** Esta función tiene la siguiente producción:
    - **Reglas:**

  Verifica que la palabra clave sea un if.

  Verifica si hay un paréntesis izquierdo (() seguido de una expresión que representa la condición del if.

  Verifica si hay un paréntesis derecho ()) después de la condición.

  Llama a parse_block() para analizar el bloque de código asociado al if.

  Si hay un else, llama a parse_block() para analizar el bloque de código asociado al else.

- **parse_while_statement():** Esta función tiene la siguiente producción:
    - **Reglas:**

  Verifica que la palabra clave sea un while.

  Verifica si hay un paréntesis izquierdo (() seguido de una expresión que representa la condición del while.

  Verifica si hay un paréntesis derecho ()) después de la condición.

  Llama a parse_block() para analizar el bloque de código asociado al while.

- **parse_for_statement():** Esta función tiene la siguiente producción:
    - **Reglas:**

  Verifica que la palabra clave sea un for.

  Verifica si hay un paréntesis izquierdo (().

  Llama a parse_declaration() para analizar la inicialización del bucle for.

  Llama a parse_expression() para analizar la condición de continuación del bucle for.

  Verifica si hay un punto y coma (;) después de la condición.

Llama a parse_expression() para analizar la expresión de actualización del bucle for.

Verifica si hay un paréntesis derecho ()) después de la expresión de actualización.

Llama a parse_block() para analizar el bloque de código asociado al for.

5. Símbolo Inicial

El símbolo inicial de una gramática en el contexto de este código sería la función parse_statement(). Esta función es el punto de entrada para el análisis sintáctico de una serie de tokens. Desde aquí, se comienza a analizar la estructura del código fuente, llamando a otras funciones según el tipo de declaración o instrucción encontrada.

La razón por la cual parse_statement() es el símbolo inicial es que se encarga de identificar el tipo de declaración o instrucción que se encuentra en el código fuente y llama a la función correspondiente para analizarla.

6. Precedencia y Asociatividad

Definir las reglas de precedencia y asociatividad para operadores:

Estas reglas determinan el orden en que se evalúan los operadores en las expresiones aritméticas y lógicas.

Operadores aritméticos (+, -, *, /):

La precedencia y la asociatividad de estos operadores generalmente siguen las convenciones matemáticas estándar. Por ejemplo, * y / tienen mayor precedencia que + y -, y la asociatividad es de izquierda a derecha.

Operadores relacionales (<, >, ==, !=, <=, >=):

Estos operadores generalmente tienen la misma precedencia, y su asociatividad es a menudo no relevante ya que se aplican a pares de valores en comparaciones.

Operadores de asignación (=):

La precedencia del operador de asignación es baja y su asociatividad es de derecha a izquierda, lo que significa que las expresiones como a = b = c se evaluarán como a = (b = c).

Otros símbolos y operadores:

Para los paréntesis (y), su precedencia es alta y se utilizan para agrupar expresiones. La asociatividad no es relevante en este caso, ya que los paréntesis siempre dictan el orden de evaluación.

Dado que el análisis sintáctico del código se centra principalmente en verificar la estructura y la sintaxis del código fuente en lugar de evaluar expresiones complejas, la definición precisa de precedencia y asociatividad para los operadores no es una preocupación principal en este contexto.

7. Comentarios y Anotaciones

```
# Función para el análisis sintáctico

def sintactico(tokens):

    errors = []

    index = 0


    # Definición de las reglas de análisis sintáctico


    # Regla general: Se continúa el análisis sintáctico hasta que se haya analizado todos los tokens.

    while index < len(tokens):

        parse_statement()  # Regla: Se llama a la función parse_statement() para analizar cada declaración o instrucción.


    return errors


def parse_statement():

    nonlocal index


    # En esta función, se analiza la estructura de las declaraciones e instrucciones en el código fuente.

    # Cada tipo de declaración o instrucción se maneja de manera específica.


    if index < len(tokens):
```

```
    line, token_type, token = tokens[index]

    # Regla: Si la palabra clave es una declaración de variable (int, float, char, long), se llama a parse_declaration().
    if token in {'int', 'float', 'char', 'long'}:
        parse_declaration()

    # Regla: Si la palabra clave es un 'if', se llama a parse_if_statement().
    elif token == 'if':
        parse_if_statement()

    # Regla: Si la palabra clave es un 'while', se llama a parse_while_statement().
    elif token == 'while':
        parse_while_statement()

    # Regla: Si la palabra clave es un 'for', se llama a parse_for_statement().
    elif token == 'for':
        parse_for_statement()

    # Si no coincide con ninguna de las anteriores, se considera una expresión y se llama a parse_expression().
    else:
        parse_expression()
```