Proyecto de Programación II HULK

Jabel Resendiz Aguirre

January 20, 2024

1 Introducción

Como parte del II proyecto de la carrera de programación de la carrera Ciencias de la Computación , se implementará un intérprete del lenguaje de programación [HULK]. Solo se definirá un subconjunto del lenguaje,definido a continuación, como parte de otro proyecto en 3er año de la carrera. En el archivo Readme.md adjunto al código se dan a conocer las exigencias para compilar el proyecto.

2 Interpreter.Logic

En este biblioteca de clases está todo la lógica del proyecto en C. Se divide en tres procesos fundamentales:

- Análisis Lexicográfico(LEXER): donde se realiza la conversión de una secuencia de caracteres a una secuencia de tokens. En esta fase se detectan los errores relacionados con la escritura incorrecta de símbolos.
- Análisis Sintáctico(PARSER): donde se determina la estructura sintáctica del programa a partir de los tokens y se obtiene una estructura intermedia. Se detectan los errores relacionado con la sintaxis(gramática).
- Análisis Semántico : donde se verifica las condiciones semánticas del programa y se valida el uso correcto de todo los símbolos definidos.

```
public class Lexer
{

public Token GetNextToken()
{

    while (CurrentChar != null)
{
    if (char.IsWhiteSpace((char)CurrentChar))
{
    SkipSpace();
    continue;
}

    if (CurrentChar == ' "\ ') // char que representa el inicio de un string "
{
    return Cadene();
}

    if (char.IsLetter((char)CurrentChar))
{
    return Variable();
}
```

```
        if(char.IsDigit((char)CurrentChar))
    {
        return Number();
    }

        switch (CurrentChar){
            case '@':
                Next();
                return new Token(TokenTypes.AT,'@');
                ...
            }
                ...
            }
        }

    }
```

2.1 Análisis Lexicográfico(LEXER)

En el código hay tres archivos que analizan este proceso. Un enum define todos los tipos de tokens posibles en el lenguaje. almacenará cada token como un tipo y su valor. Además se define una clase ReservedKeyword que contiene todas las palabras reservadas del lenguaje, entiendase por reservada que su uso en el lenguaje define la invocacion de una instruccion o parte importante de la sintaxis del lenguaje. Evite usar estas palabras como parte de nombres de variables. Una clase Lexer que buscará dentro de la cadena el próximo token de la cadena mediante la siguiente sintaxis: Este método se encargará de pasar char por char de la cadena incial y encontrar con métodos auxiliares la forma de encontrar que tipo de token es.

Cuestiones generales

- Los errores léxicos se detectan por char que no encontramos válidos en el lenguaje
- Además por mal uso de los tipos, osea combinaciones de dígitos y letras

```
    program : statement_list

    statement_list : compounds

    statement : declarations
              | assignment
              | conditionals
              | functions
              | call_functions
              | print
              | compounds

    declarations : LET + ID (COMMA ID)* + EQUAL + compounds + IN + compounds

    assignment : ID + ASSIGN + compounds

    conditionals : IF + L_PARENT + compounds + R_PARENT (RETURN)* + compounds + ELSE (RETURN)*+ compounds

    functions :FUNCTIONS + ID + L_PARENT + ID (COMMA ID)* + R_PARENT + RETURN + compounds

    call_functions:  L_PARENT + compounds (COMMA compounds)* + R_PARENT

    print: PRINT + L_PARENT + compounds + R_PARENT

    compounds : comp + ((AND | OR) comp)*

    comp : expr + ((SAME | DIFFERENT | LESS | GREATER | LESS_EQUAL | GREATER_EQUAL | NOT ) expr)*

    expr : term + ((PLUS | MINUS | AT ) term)*

    term : exp + ((MUL | DIV | MOD) exp)*

    exp: factor + ((POW) factor)*

    factor :statement
           | PLUS factor
           | MINUS factor
           | NUMBER
```

```
            | STRING
            | BOOLEAN
            | PI
            | ID
            | TRUE
            | FALSE
            | L_PARENT compounds R_PARENT


    type_data :
               NUMBER
             | BOOL
             | STRING
```

2.2 Análisis Sintáctico(PARSER)

Este proceso sigue una grámatica específica. El proceso en la clase Parser analiza la cadena para producir un árbol de derivación final. Cumpliendo la jerarquía, se crea una lista de instrucciones que reconoce la instrucción que existe en la línea. El algoritmo a pensar es dado un token , saber si es terminal o no. Símbolos terminales son aquellos que estarán en el método Factor.

Se sigue un análisis recursivo para crear cada nodo nuevo del árbol. Muchos token solo forman parte de la gramática específica de la instancia por tanto su uso cumple una regla.

Cuestiones generales

- Los errores sintácticos son detectados por tokens inesperados, osea que un token forme parte de una grámatica específica y no se encuentre por ejemplo: después de un print le sucede un parentésis abierto, si al poner print no se encuentra tal paréntesis pues se devolverún error
- Tambiéncontrará token que no invocan una instrucción como poner : (in print(23)), el cual el token IN no invoca una instrucción y por tanto se usa mal.

```
using System;

public abstract class NodeVisitor

{

protected object Visit(AST node,Dictionary<string,object> some)
{
            ...
        if (node is BinaryOperator)

        return VisitBinaryOperator((BinaryOperator)node,some);

        else if (node is Instructions)

        return VisitInstructions((Instructions)node,some);

        else if (node is Num)

        return VisitDeclarations((Num)node,some);
            ...
}

        ...
public abstract object VisitBinaryOperator(BinaryOperator node,Dictionary<string,object>Scope);
public abstract object VisitInstructions(Instructions node,Dictionary<string,object>Scope);
public abstract object Num(Num node,Dictionary<string,object>Scope);
        ...
}
```

```
    public class Interpreter : NodeVisitor
{
 ...
```

```
    public Interpreter(Parser parser){
       ...
    }
    public override object VisitBinaryOperator(BinaryOperator node,Dictionary<string,object>Scope)
    {
        object result = 0;

        object left = Visit(node.Left,Scope);
        object right = Visit(node.Right,Scope);

        switch (node.Operator.Type)
        {
           case TokenTypes.PLUS:
               ...
               if(left is string)
                   result= (string)left + (string)right;
               else
                   result=Convert.ToDouble(left)+ Convert.ToDouble(right);
               break;
           ...
        }
    }


    public override object VisitInstructions(Instructions node,Dictionary<string,object>Scope)
    {
        foreach (var item in node.Commands)
        {
            object output =Visit(item,Scope);
            Console.WriteLine((output is string)?(string)output: (output is bool)? (bool)output :
                 Convert.ToDouble(output));
            Scope.Clear();

        }

        return 0;
    }

    public override object Num(Num node,Dictionary<string,object>Scope)
    {
       return node.Value;
    }
    ...
   }
```

2.3 Análisis Semántico

En esta fase se evaluará el AST, evaluando el subárbol izquierdo y después el subárbol derecho. Recursivamente se continúa de esta forma por cada nodo del árbol hasta llegar a una hoja(nodo que no contiene hijos). Cada hoja representará un símbolo terminal y por tanto tendrá asociado un valor, el cual será devuelto. Veamos un ejemplo código

La clase NodeVisitor está implementada para en su método Visitor se reciba un nodo , se localice de que tipo es(en este caso Declaration, Instruccion o BinaryOperator) para acceder entonces al método que lo implementa. La clase Interpreter que hereda de NodeVisitor debe implementar cada método abstracto de su clase padre.

Como notamos el método BinaryOperator al recibir un objeto BinaryOperator que contiene campo su rama izquierda, una para la derecha y un campo con el signo guardado. Evalúa cada nodo (derecho e izquierdo) y realizará la operación entre el valor de cada nodo dado su signo.

Por otro lado el método Instruccion al recibir un objeto Instruccion que contiene un campo con una lista de AST y evaluará cada uno de ellos. Aunque dado las características de este lenguaje cada línea representará un AST.

El método Num al recibir un objeto Num que contiene un campo con el valor del número , lo retorna tal cual.

De esta forma y análogamente para cada método se evalúa recursivamente el árbol.

Cuestiones generales

- Toda línea imprime un resultado que será el valor del árbol creado, incluso la declaración de funciones en el lenguaje en el cual se retorna 0 por defecto y muestra de que se ha incluido correctamente

- Los errores semánticos son obtenidos de errores durante la evaluación, ya sea por realizar operaciones entre objetos de tipo diferente sin usar los símbolos permitidos,por obtener valores de excepción del compilador de C#.

3 Interpreter.Visual

Se ha creado en la biblioteca dos clases como parte de la interfaz gráfica de la app de consola. No es muy avanzado pero incluye un menu de opciones, "PLAY" para iniciar con el compilador , "ABOUT" que dará más información sobre el intérprete y cómo funciona ,y una última opción "EXIT" que dará fin al programa y lo cerrará. En estas clases se lleva un contador que aumenta(o disminuye) a través de la matriz de opciones,dependiendo de si se usan las teclas arriba o abajo. Se limpia la consola con el método Clean() y se restablece el diseño.Incluye además un menú para definir el color.