

Segundo Proyecto de Programación Matcom

Primer Año

José Miguel Leyva De la Cruz

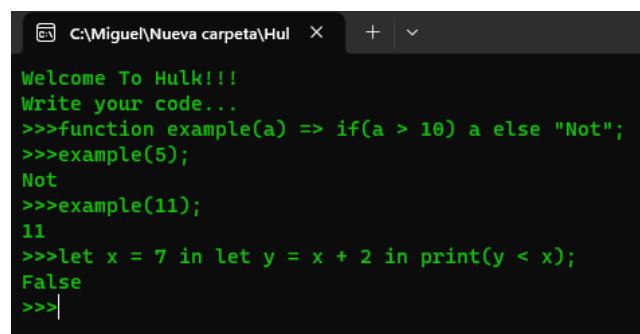
Introducción

¿Qué es un Compilador? Un compilador es un programa informático que traduce todo el código fuente de un proyecto de software a código máquina antes de ejecutarlo. Básicamente, recibe una serie de instrucciones (comandos o programas), las cuales analiza y procesa para posteriormente ejecutarlas.

Havana University Language by Kompilers o simplemente HULK, es el lenguaje que se desea Compilar con este trabajo computacional. Para llevar a cabo la acción de Compilar dicho lenguaje, el programa hace su curso por cuatro etapas fundamentales: Análisis Léxico, Análisis Sintáctico, Análisis Semántico y Evaluación de Código. Todas las instrucciones recibidas por esta versión de HULK serán instrucciones **inline**, o lo que es lo mismo: se expresan en una línea.

La Aplicación

La aplicación consiste en una aplicación de Consola, con letras verdes. Su funcionamiento es muy sencillo, el usuario introduce la instrucción deseada, al pulsar la tecla *ENTER* se visualizará inmediatamente en la siguiente línea, la evaluación de su entrada (en caso de ser una declaración de función, esta simplemente se guardará para futuros llamados). El proceso se continuará repitiendo, hasta que el usuario presione la tecla de *ESCAPE*, lo que provocará el cierre de la aplicación.

A screenshot of a console window titled 'C:\Miguel\Nueva carpeta\Hul'. The window has a dark background with green text. The text inside shows a welcome message, a prompt to write code, and several lines of code being executed with their outputs. The code includes a function definition, a function call, and a conditional statement.

```
Welcome To Hulk!!!
Write your code...
>>>function example(a) => if(a > 10) a else "Not";
>>>example(5);
Not
>>>example(11);
11
>>>let x = 7 in let y = x + 2 in print(y < x);
False
>>>|
```

Figure 1: Ejemplo de Código en Hulk

Etapas del Proceso de Compilación

1 Análisis Léxico(Tokenización)

Un **Token** no es más que una secuencia de caracteres, los cuales forman una cadena, que juntos cumplen ciertas características ya predefinidas. Existen varios tipos de **Tokens**, como son: las palabras claves de algún lenguaje (int, string, for, if), los nombres de las variables o las funciones (a, x, fib), los Operadores aritméticos, de relación y lógicos (+, >, =, &) y finalmente los literales numéricos, cadenas o booleanos.

La aplicación utiliza la clase Token, la cual define al objeto Token. En su campo se encuentra un enum **TokenType** almacenando todos los tipos de Tokens que utiliza el programa (La utilización del enum se debe a que si se definen los tipos de Token mediante strings, es más probable equivocarse al escribir un tipo de Token en alguna ocasión). Además emplea un diccionario para guardar todos aquellos **Tokens** que están prefijados en el lenguaje. El constructor de esta clase recibe un objeto de tipo **TokenType** y un string que representa el valor del Token.

La clase estática **Tokenizer** posee el método GetTokens, el cual recibe la línea de código introducida por el usuario y realiza todo el proceso de Tokenización de la misma, preguntando a cada carácter si es inicio, continuación o fin de Token. Una vez leída y analizada la entrada, el método devuelve la lista de Tokens correspondiente.

2 Análisis Sintáctico (Parser)

En esta etapa, el Compilador obtiene la lista de Tokens previamente creada y procede a determinar si dicha lista (secuencia), es correcta. La definición de una secuencia de Tokens correcta puede variar en dependencia del lenguaje, en caso de HULK algunas secuencias incorrectas serían: una expresión binaria que solo tenga un miembro, una expresión no balanceada en cuanto a paréntesis se refiere y otros casos. Para ello emplea una gramática libre de contexto, en la cual está reflejada la sintaxis de HULK. La gramática empleada es la siguiente:

```
L = M;  
M = let id = MH | if M else M | print(M) | function id (Q) => M | A  
H = in M | e  
Q = id | id , Q | e  
K = E | E , K | e  
A = BZ | !BZ  
Z = and A | or A | e  
B = EW  
W = < E | > E | <= E | >= E | ==E | != E | = E | e  
E = FX  
X = + E | - E | e  
F = TY  
Y = * F | / F | ^ F | e  
T = int | (M) | true | false | id | func(K)
```

Figure 2: Gramática

Esta etapa no solo analiza si la secuencia de Tokens cumplen con la Sintaxis de HULK, además genera el Árbol de Sintaxis Abstracta (AST según sus siglas en Inglés). En este AST, los nodos son las operaciones que se llevarán a cabo

y los hijos son los operandos, estos a su vez pueden estar compuestos por otras operaciones. Con el AST se logra definir la instrucción que se le dará al Compilador de una manera recursiva.

Para lograr generar el AST, se emplea la clase abstracta **Expression**. Al utilizarse el modificador **abstract** se tiene la ventaja de que si se crea un nuevo objeto que herede de Expression, este podrá implementar todos sus métodos de forma independiente, en dependencia del tipo de expresión y de las necesidades del nuevo objeto.

3 Análisis Semántico

Una vez creado el AST, durante el análisis semántico se desea saber si todas las operaciones están trabajando con los operandos correctos, por ejemplo si la expresión de suma, está trabajando con dos números o con dos expresiones que devuelvan un número o con dos variables que tienen como valor un número. Es un trabajo bastante complejo, pero gracias a la definición recursiva de la expresión (AST) se puede realizar de una forma no muy complicada.

Teniendo en cuenta que se está trabajando con la estructura de datos conocida como Árbol, la forma más cómoda de operar con dicha estructura es recorriéndolo. El recorrido que se emplea en esta ocasión es en pre-orden, es decir, partiendo desde la Raíz. El análisis Semántico realiza dos recorridos:

- El primero para crear el **Scope** de la expresión, así como los **Scopes** de las expresiones hijas, en caso de ser necesario.
- El segundo para verificar si las operaciones están utilizando el tipo de operando correspondiente

La creación de los Scopes se basa en determinar en qué ámbito fue declarado una variable y en cuál será usado. La definición de Scope para esta aplicación, consiste en una estructura enlazable muy similar a la LinkedList (Estructura del lenguaje C#), en la cual, cada Scope tiene acceso al Scope, valga la redundancia, que está en el nivel inmediato superior. De esta forma, si se quiere saber qué valor posee una variable, basta con recorrer los Scopes de los niveles superiores en donde esta se está usando.

El chequeo de tipos, es el penúltimo paso que se realiza antes de proceder con la evaluación de la expresión. El Scope de la expresión ya está creado y tiene almacenados todas las variables declaradas y los tipos de datos que almacenan dichas variables. Evidentemente, la solución al problema de chequear si una expresión de suma opera con dos números es, como bien se mencionó anteriormente, recorrer el AST una vez más. Durante el recorrido cada vez que se se analice una operación, se pedirá el tipo de datos que representan los hijos de dicha expresión, si son los correctos, el recorrido continuará.

4 Evaluación

Una vez analizada léxica, sintáctica y semánticamente la entrada del usuario, se procede con la evaluación de la misma y se devuelve el resultado correspondiente.

Otros Detalles

Sobre los Errores

Existen tres tipos de errores, en el lenguaje HULK, estos serían: Error Léxico, Error Sintáctico y Error Semántico. Evidentemente el nombre de cada error, da pista con respecto a en que momento del proceso podrían ser lanzados.

- Error Léxico: Como bien indica su nombre, aparece durante la etapa de análisis léxico y normalmente se debe a expresiones que contengan tokens inválidos (14a, "Hola).
- Error Sintáctico: Durante el proceso de Parser, como se mencionó anteriormente, se revisa si una secuencia de tokens es válida o no, en caso de no serlo a parece este error. Algunos ejemplos de cadenas inválidas serían: `if(+ 5)`, `(8 - 5) * 3)`, `@ "example"`.
- Error Semántico: Este error puede encontrarse en dos ocasiones. La primera sería durante la creación del Scope, en donde se revisa si se utiliza una variable previamente creada o se llama alguna función almacenada en la memoria. La segunda ocasión es durante el chequeo de tipos, donde se determina si las operaciones están usando los operandos correspondientes, un ejemplo de una sentencia inválida sería: `80 + "Azul" ó "Rojo" * true`.

Sobre la clase Expression

La clase abstracta Expression, es la clase fundamental de la aplicación. El método GetScope, es quien recorre el AST (quien en este caso es la expresión devuelta por el Parser) y crea los scopes correspondientes, nótese que cada tipo que herede de Expression, rellenará el Scope actual que es pasado como parámetro. El método SemanticWalk, procede con el chequeo de tipos una vez creado el Scope en el método antes mencionado. Finalmente Evaluate realiza la acción de evaluar la entrada del usuario y devolver el resultado. Recaltar que al ser Expression una clase abstracta todos los tipos que hereden de esta, tendrán su propia implementación de estos métodos.

Conclusiones

La aplicación presentada, se considera un compilador de una versión minimalista del lenguaje HULK. Sin embargo está capacitada para leer y realizar operaciones con instrucciones que posean cualquier tipo de operación aritmética, condicionales, declaración de funciones, llamado de funciones e incluso la capacidad de evaluar funciones con implementación recursiva.

```

namespace HULK_COMPILER
{
    151 references
    public abstract class Expression
    {
        47 references
        public abstract void GetScope(Scope actual);
        45 references
        public abstract Scope.Declared Semantic_Walk();
        48 references
        public abstract string Evaluate();
    }
}

```

Figure 3: Expression

Bibliografía Consultada

1. Compylers by Piad: Libro utilizado para el estudio de la asignatura Compilación. Autor: DrC. Alejandro Piad Morffis, profesor de la Facultad de Matemática y Computación de la Universidad de La Habana.
2. Artículos de Wikipedia en idioma inglés que abordan temas de Análisis Léxico, Sintáctico y Semántico