IIProyecto de Programacion I.Curso 2023

Facultad de Matemática y Computación

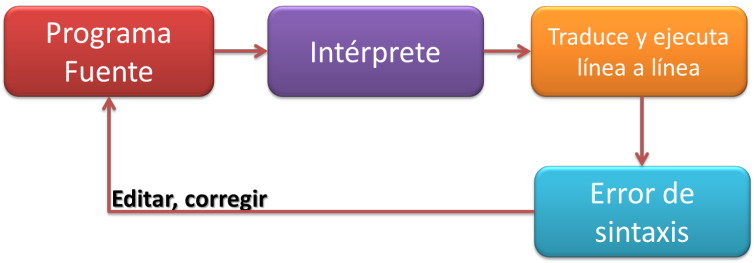
Javier A. González Díaz. C-113

Resumen

Los intérpretes son herramientas esenciales en el mundo de la programación. Son programas que leen y ejecutan otros programas, traduciendo cada instrucción del código fuente a medida que se necesita Esto contrasta con los compiladores, que traducen todo el programa a código de máquina antes de su ejecución.

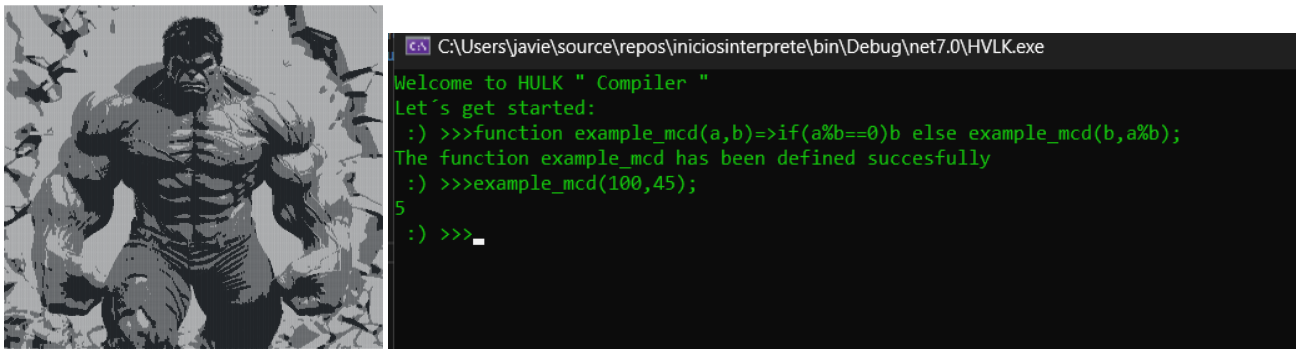
Aunque los programas interpretados pueden ser más lentos que los compilados, ofrecen una mayor flexibilidad. Los intérpretes facilitan la modificación y depuración del código, permitiendo reemplazar partes del programa o añadir módulos nuevos sin necesidad de recompilar todo el código.

Además, los intérpretes proporcionan un entorno de ejecución independiente de la máquina física, conocido como máquina virtual. Esto significa que un programa interpretado puede producir los mismos resultados en diferentes sistemas, siempre que se ejecute con el mismo intérprete.



Aplicación:

En este proyecto desarrollare un intérprete para una version simplificada del lenguaje HULK(Havana University Language for Kompilers).Nuestra aplicación tiene un funcionamiento simple , es una aplicación de consola que recibe instrucciones *inline* y cuando el usuario presiona *ENTER* se muestra en la siguiente línea el resultado de la ejecucion y evaluación de la instrucción .El proceso es cíclico y guarda definiciones de funciones para su poterior manejo hasta que el usuario desee cerrar la aplicación.



Ventajas de la interpretacion de código:

1. Su principal ventaja es que permiten una fácil depuración. Permiten una mayor interactividad con el código en tiempo de desarrollo.

2. Un Intérprete necesita menos memoria que un compilador.Puede ser rápidamente modificado y ejecutado nuevamente.

3. Portabilidad
4. Un intérprete siempre procesa el código línea por línea, de modo que lee, analiza y prepara cada secuencia de forma consecutiva para el procesador. Este principio también se aplica a las secuencias recurrentes, que se ejecutan de nuevo cada vez que vuelven a aparecer en el código
5. Facilita la búsqueda de errores, ya que la línea de código problemática se detecta inmediatamente después de ocurrir el fallo.

Algoritmo que sigue nuestro Intérprete:

1. *Análisis Lexicográfico:*

El análisis léxico es la primera fase de un compilador, y también se aplica en el proceso de interpretación. Este proceso toma el programa fuente, que está escrito en forma de declaraciones, realmente está formada por una secuencia de caracteres, y lo desglosa en una serie de tokens.

Un token es la secuencia de caracteres agrupadas como una unidad lógica. Estos tokens son los componentes léxicos o símbolos que se producen como salida del analizador léxico. El intérprete realiza la función de traducir el programa con una declaración a la vez. Durante este proceso, se eliminan todos los comentarios en el código y los espacios en blanco. Además se especifican el tipo de token correspondiente a es segmento de la cadena de entrada y su posicion en ella para el posterior manejo de errores.

1.1. Anatomía de nuestro *Lexer*

Almacena el texto de entrada y los términos que aparecen en la misma.

Realiza el recorrido caractér a caractér ,auxiliándose de métodos que analizan qué representa :un número , una letra , un símbolo u otro permitido por nuestro lenguaje , y de esa forma llamar a métodos como :

- Extract Number()
- Extract String()
- Extract identifier()
- Extract Operator()

Y así de forma análoga para cada uno de los tipos que tiene el lenguaje y de esta forma poder completar los tokens según el primer caractér o el prefijo de cadena ya reconocido , siempre adhiriéndonos a las reglas lexicales del lenguaje.De esta forma se logra la detección temprana de errores y que la entrada adquiera una lógica semántica aún mayor para el siguiente proceso.

2. *Análisis Sintáctico:*

2.1. Funciones de nuestro *Parser*

Es la fase del analizador que se encarga de chequear el texto de entrada en base a una gramática libre de contexto dada.

```
//Gramática libre de contexto
//Program P-> S_L
//Statement_list S_L -> S | S ; S_L
//Stat S -> l_v|d_f|p_s|if (Bo) else | E
//Bo Boolean Bo-> EWZ| !EWZ  | EW | !EW
//W -> < E | > E | <= E | >= E | ==E | !=E
//Z-> and Bo | or Bo
//Argument List A_L-> id | id "," | E
//Expression E-> T+E | T-E | T
//Termino T-> F * T | F / T | F % T | F^T | F
//Factor F -> -F | (E) | A
//Atom A-> number|ID|func-call
```

Y en caso de que el programa de entrada sea válido, suministra el árbol sintáctico que lo reconoce. En teoría, se supone que la salida del analizador sintáctico es alguna representación del árbol sintáctico que reconoce de la secuencia de tokens suministrada por el analizador léxico.

En la práctica, el analizador sintáctico también hace:

- Acceder a la tabla de símbolos (para hacer parte del trabajo del analizador semántico).
- Chequeo de tipos (del analizador semántico).
- Generar el árbol sintactico que represeta la entrada en caso de ser válida
- Generar errores cuando se producen.

En definitiva, realiza casi todas las operaciones.Además garantiza captar errores como parentesis no balanceados , expresiones binarias incompletas ,etc.

2.2. *Árbol de sintaxis :*

El árbol de derivación no nos da todas las facilidades para el próximo proceso de chequeo de semántica y evaluación por lo tanto creamos un árbol de sintáxis. En especifico para este interprete creamos una clase para cada expresión que interviene en el lenguaje, ejemplo:

- Nodo : como clase que representa cada uno de los nodos del arbol que se va produciendo
- Instrucciones como declaraciones de variables, expresiones condicionales ïf-else", definiciones de funciones y uso de otras ya predefinidas
- En un nivel más bajo encontramos las expresiones unarias y binarias usadas tanto para expresiones aritmeticas , con una jerarquia bien definida para respetar la precedencia operacional, como para expresiones booleanas
- Expresiones atómicas como números , identificadores de variable , llamados a funciones ,cadenas de texto , etc.

Cada tipo de expresión bien representada con el fin de lograr una diferenciación a la hora de la evaluación y validación semántica.

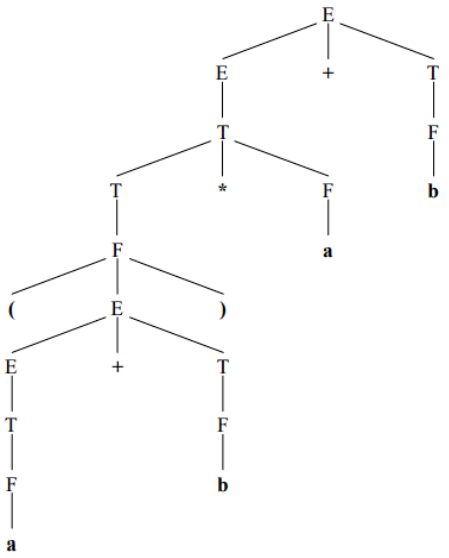
En específico la clase absract EXPRESSION permite aprovechar la herencia para ese trato distintivo en la evaluación y ese orden que debe seguir el Parser para construir el árbol.

```
3 referencias | Javi111003, Hace 4 días | 1 autor, 1 cambio
public abstract class Node
{
    protected List<Node> children=new List<Node>();

}
64 referencias | Javi111003, Hace 4 días | 1 autor, 1 cambio
public abstract class Expression : Node
{
    60 referencias | Javi111003, Hace 4 días | 1 autor, 1 cambio
    public abstract object Evaluate();

}
```

Este es un ejemplo bien sencillo de un arbol de sintaxis que se crea como resultado del parsing recursivo de una expresion aritmética:



3. *Análisis Semántico :*

3.1. Acciones durante el chequeo semántico:

1. Se chequean los tipos de las expresiones y en donde se usan para verificar su correcto empleo
2. En el uso de funciones se espera que esta haya sido definida con anterioridad(al igual que las variables) y que todos sus argumentos hayan sido proporcionados para la evaluacion
3. Maneja lo referido al chequeo del *CONTEXTO*

3.2. Contexto:

El CONTEXTO también conocido como tabla de símbolos es donde se almacena la informacion de variables y funciones para su posterior uso. En el caso de este intérprete tiene soporte para redefinir variables , al igual que una implementacion conveniente con listas de tuplas y evaluaciones de variables en orden inverso simulando el comportanmiento de los Scopes o mini contextos que se crean dentro de cada llamado a una función, para soportar la implementacion de recursividad,así aportando una mayor utilidad y eficiencia.

4. *Evaluación:*

4.1. Explicación

Cuando una entrada pasa por todas las fases anteriores ya está validada y se procede con la evaluación , cuyo resultado depende de la expresión invocada , a la cual ya se le ha predefinido desde la construccion de ella , el modo en el que se evalúa y el tipo que retorna.

5. *Manejo de errores:*

Para esto se ha creado una clase error que deriva de la clase Expression pudiendo así ser devueltas por cada uno de los componentes del intérprete , estos se almacenan en una lista para poder guardar la mayor parte de errores durante la ejecución, esta se ubica en la clase principal de la App:*Program* Al finalizar cada fase del proceso se chequea que esta esté vacia la lista para proseguir ,en caso de que contenga errores se interrumpe de inmediato el proceso , explicando :

- Tipo de error(Léxico,Sintáctico o Semántico)
- Mensaje especifícatico del error

- Ubicación en el código fuente del error(Columna,ya que las instrucciones son *inline*)

Conclusiones:

Se seguirá trabajando en la optimizacion y mejoramiento de dicho proyecto con la intención de mejorar siempre la calidad de respuesta a las necesidades del usuario.

Índice