Facultad de Matemática y Computación

TERCER PROYECTO DE PROGRAMACIÓN

Colaboladores:

Joel Aparicio Tamayo
Claudia Hernández Pérez

Grupo: C-113

Lenguaje G-Sharp

Estructura:

La implementación del intérprete del lenguaje G# se ha realizado en un proyecto de tipo *Class Library* y consta de un conjunto de archivos en lenguaje C# que agrupan toda la lógica. Posee un *Lexer*, un *Parser*, un *Evaluator*, un *Checker*, y por supuesto toda la sintaxis de cada expressión del lenguaje. Las expresiones soportadas en el lenguaje son:

- **UnaryExpressions**: Expresiones unarias con los operadores +, -, not.

- **BinaryExpressions**: Expresiones binarias con los operadores $+, -, *, /, \%, and, or, <, <=, >, >=, ==, !=$.

- **UnaryExpressions**: Expresiones unarias con los operadores +, -, not.

En la carpeta *Types* se encuentran:

- **Boolean:** Es la clase que evalúa los tipos booleanos.
- **Function:** Es la clase que evalúa las funciones predefinidas. • **Numeric:** Es la clase que evalúa los tipos numéricos.
- **Parenthesis:** Es la clase que evalúa los paréntesi.
- **String:** Es la clase que evalúa los strings.
- **Types:** Es la clase madre de la carpeta que dtermina si la entrada es de un tipo básico y lo evalúa.

Además existe una carpeta interna *Operations* que contiene 3 archivos que agrupan por tipos las evaluaciones a realizar.

Fuera de *Expressions* se encuentra la carpeta *Engine* que posee 5 clases:

- **Cache:** Es la clase que guarda los datos del programa.
- **Check:** Es la clase que revisa la gran mayoría de errores sintánticos y semánticos.
- **Extra:** Es la clase que contiene métodos auxiliares para el programa.
- **Main:** Es la clase principal de la biblioteca de clases.

Funcionamiento

- **Tipos Básicos:**

Considerando una entrada con tipos básicos, dígase operación aritmética, booleana o un string, la entrada va a pasar por la clase `Main` y ahí se comprueba primeramente que no tenga errores de sintaxis. En caso de ser correcta, se llama a la clase `Types` donde se determinará exactamente qué tipo es la expresión entrante de manera global, y luego recursivamente se llamará para cada parte de la misma. Aquí se decide qué hacer con la expresión. En orden de prioridad:

- si contiene paréntesis se evalúa en la clase `Parenthesis`

- si es string se evalúa en la clase `String` si es un string o hay que realizar alguna concatenación.

- si es booleano se evalúa en la clase `Boolean` qué tipo de operación booleana hay que ejecutar o si es directamente 'true' o 'false'.

- si es numérico se evalúa en la clase `Numeric` qué tipo de operación numérica hay que ejecutar o si es directamente un número.

En cada una de las operaciones se llama recursivo al método `Parse` de la clase `Main` para cada miembro de la operación. Y entonces vuelve a ejecutarse el programa. Así con cada parte de la expresión global.

Ejemplo: (1 + 2) * 3 == 9;

La entrada pasará primero por la clase `Main`. Allí se revisa que los paréntesis estén balanceados y que la expresión termine en ';'. Una vez pasa estas revisiones se llama al método `Parse`. Primeramente pasará nuevamente por un par de filtros, uno sintáctico(`SyntaxCheck`) y uno semántico(`SemanticCheck`), ambos en la clase `Check`.

Como la expresión es totalmente válida pasará los filtros y entonces será evaluada en la clase `Types`. Por prioridad primero se evaluarán los paréntesis en la clase `Parenthesis`. Dicha evaluación consiste en buscar los índices del último paréntesis abierto y el primero cerrado a partir de este (que será su complementario) y analizar la expresión contenida entre ambos con el método `Parse` de la clase `Main`. En este caso procederá a analizar '1 + 2', que hará el mismo recorrido hasta llegar a `Types`, pero esta vez al no contener paréntesis, lo evaluará como una expresión numerica mediante la clase `Numeric`.

Aquí se buscará cuál es la operación a realizar. En este caso la suma (+):

Se llamará a la subclase `Add` que evaluará dicha suma y retornará '3':

Los métodos van devolviendo recursivo hasta llegar a `Parenthesis`, donde se habrá susituido el espacio ocupado por la expresión parentetizada, por su valor de retorno:

A continuación se retorna nuevamente el análisis de la expresión global '3 * 3 == 9'. Vuelve a entrar a `Types`, pero esta vez la prioridad hace que entre en `Boolean` donde se evalúa la igualdad (==). Para ello primero se evalúa cada miembro por separado. El MD después de analizarlo sabemos directamente que devolverá '9'. Pero el MI, al igual que el '1 + 2' va a ser evaluado eventualmente en `Numeric` pero esta vez como multiplicación (*). Igualmente se evalúa cada miembro por separado, lo cual se realiza sin complejidades porque ambos son números, así que su valor de retorno son ellos mismos. Luego se efectúa la multipliación en la subclase `Mult` retornando '9':

Así, va devolviendo el valor recursivamente hasta llegar a `Boolean`, donde efectuará la operación de igualdad en la subclase `Equal` retornando 'true':

Este valor se retorna recursivamente hasta `Parse` en `Main`, y de aquí se retorna hasta la salida en `Program`, imprimiendo 'true':

- **Cálculo de funciones:**

Si consideramos ahora una entrada con funciones ya creadas, va a pasar igualmente por la clase `Main` y ahí se comprueba primeramente que no tenga errores de sintaxis. En caso de ser correcta, se llama a la clase `Types`, donde por la existencia de paréntesis en la función, se evaluará en la clase `Parenthesis`.

En esta clase se llama al método `GetData` de la clase `Function`. Este método devuelve una tupla de tres elementos. El primero es un valor booleano que indica si los paréntesis dados pertenecen a una función. En caso que no, el programa continúa la ejecución tal y como en el ejemplo de **Tipos Básicos**. En caso que sí sea una función entonces se guardan los otros dos valores de retorno de `GetData`: un string con el nombre de la función y un array con los argumentos. Luego, con esos datos se evalúa con el método `Eval` de la clase `Function`. Finalmente se sustituye el valor retornado en la expresión global y se retorna su evaluación en `Parse` de `Main`.

Ejemplo: (1 + 2) * 3 == 9 + sin(0);

Este ejemplo, al igual que el anterior entrará en la clase `Parenthesis`. Pero esta vez, como el último paréntesis abierto es el que está después de 'sin', será ese el que primero se evalúa. Por tanto, siguiendo el recorrido definido anteriormente, se llama al método `GetData`. Aquí, una vez entrados los índices, se procede a buscar hacia la izquierda de '(' hasta donde se encuentre un caracter que no sea dígito, letra o guión bajo. De esa forma se obtiene el nombre de la función. En este caso sería 'sin(' (se toma el paréntesis). Luego se guardan los argumentos(todo lo que está entre los paréntesis) separando cada uno por las comas. En este caso el único argumento sería '0':

Al regresar a `Parenthesis` se llama al `Eval` de `Function`. Aquí se crea un objeto 'result' de tipo 'Function' que lo que hará será guardar los valores de la funciones predefinidas evaluadas en los argumentos pasados.

En este caso se evalúan en '0'. Luego se decide cuál es la que se necesita('cos') y se retorna el valor('0'). Al retornar a `Parenthesis` se sustituye en la expresión global quedando: '(1 + 2) * 3 == 9 + 0'. Y ya esta expresión se resolverá como lo visto con anterioridad en **Tipos Básicos**.

• **Instrucción 'function':**

Una declaración de funciones entrará, como todo, en `Main.Parse`, pero esta vez entrará en `Instructions`. Aquí se evaluará como instrucción 'function' llamando al método `CreateFunction` de la clase `FuncInstruction`. En dicho método se hace una evaluación de cada parte, dígase nombre correcto, argumentos válidos, estructura correcta, errores semánticos en el body..., todo llamando a `FunctionRevision` de `Check`. Si la declaración pasó cada una de las revisiones, entonces está lista para 'crearse'. Esta terminología implica establecer mediante diccionarios y listas, una relación 'nombre-lista de variables', 'nombre-body', 'nombre-valores de entrada', 'nombre-valor de retorno'. Luego se retorna vacío, pues 'no existe' para la declaración 'function' un valor de retorno.

Ejemplo: function f(x) => x + 1;

Como en la explicación, la declaración entra en `FunctionRevision`. Primeramente se revisa que la función tenga al menos un nombre, paréntesis, y '=>'. Luego se revisan las especificidades:
- nombre válido.
- argumentos faltantes, repetidos, o simplemente que son variables inválidas.
- Body vacío.
- Sintaxis del body(tokens inválidos, variables no definidas).
- Semántica del body(operaciones correctas).

En el ejemplo la declaración es totalmente válida. Así que pasará todas estas revisiones. Una cosa importante a destacar es que en medio de las revisiones se detectará cuál es el valor de entrada que recibirá 'f' una vez creada, pues la 'x' se utiliza en una suma, por lo tanto tiene que ser número. Este mismo análisis se realiza en la revisión semántica. Por tanto, se procede a guardar los datos de 'f':
- cuerpo: ['f'] = 'x + 1'
- variables: ['f'] = $'\{x\}'$

- valor que recibe: ['f'] = $'\{number\}'$

- valor que retorna: ['f'] = 'number' (obteniendo el tipo del cuerpo con `Types.GetFunctionType`)

• **Instrucción 'let-in':**

Una instrucción let-in se evalúa en `Main.Parse` y luego pasa a la clase `Instructions`, donde se decide qué tipo de instrucción se requiere evaluar. En este caso, se evaluará como 'let-in' en la clase *Let_in*. Aquí, primeramente se revisa que no exista nada inválido antes del inicio de la declaración. Luego se revisa que contenga el token 'in'. A continuación, se revisan las variables y sus respectivos valores, y se guardan en listas, que se usan más tarde para la sustitución en el cuerpo del let-in. Dicha sustitución la realiza el método `Eval` de la clase `FuncInstruction`.

Este método hace split en el cuerpo por una serie de símbolos. Una vez separado el cuerpo, se copia cada elemento del primer array para otro nuevo, a excepción de las variables definidas, de las cuales lo que se pasa es su valor entre paréntesis. Luego se busca con un puntero de atrás hacia adelante en el cuerpo las expresiones del primer array y se sustituye por su equivalente en el segundo (la misma expresión menos las variables):

Ejemplo: let x = 1 in (x + 2) * 3 == 9

Como lo dicho anteriormente, la entrada pasa por la clase *Let_in*. Esta entrada pasará las primeras revisiones y se procede a obtener las variables y los valores. Para ello, se separa la porción de la entrada que se encuentra entre el 'let' y el 'in' por las comas que tiene, en este caso 'x = 1' es el único elemento que queda. Luego se separa por el '=' y se obtiene como variable 'x' y como valor '1'. Ninguno de los dos elementos contiene errores, así que pasarán las revisiones.

En la evaluación en `FuncInstruction` se hace split por los elementos que no sean letras, números o '_', quedando {x, 2, 3, 9}. Luego se crea un nuevo array de la misma longitud donde se van a copiar todos los elementos de este, a diferencia de la variable, que queda {(1), 2, 3, 9}

Luego se sustituye en el cuerpo cada uno de los elementos del array que contiene los cambios quedando '((1) + 2) * 3 == 9':

Este cuerpo es retornado a *Let_in* donde se retorna su evaluación en `Main.Parse`, la cual ya se ha visto como se resuelve.

- **Instrucción 'if-else':**

Una instrucción 'if-else' se evalúa en `Main.Parse` y luego en `Instructions`. Esta vez entrará en la clase `Conditional`. Aquí se realizan revisiones análogas a las hechas en *Let_in*(nada inválido antes del 'if', existencia del 'else'). Luego se llama al método `GetData` de esta misma clase y se obtienen los datos de la condicional: condición, cuerpo 1, cuerpo 2, índice donde inicia la declaración e índice donde termina.

Con estos datos se procede a evaluar la condición. Si es 'true' entonces el retorno de la expresión será el cuerpo 1, sino el cuerpo 2(cuerpo que sucede a un token 'elif' o 'else'). En caso de que exista un 'elif', el cuerpo 2 será otra condicional(nótese que al picar la palabra 'elif' por la mitad se obtiene 'if'. Luego el cuerpo 2 sería desde ese índice hasta el 'stop' marcado con anterioridad) que se resuelve de la misma forma. En otro caso, se devolvería una expresión de las anteriormente mostradas

Ejemplo if(3 < 4) (1 + 2) * 3 == 9 else 0

En este ejemplo se llegará a la clase `Conditional` y se llamará a `GetData`. El método hace algunas revisiones básicas y luego procede a ir obteniendo datos. El índice de inicio se marcará en este caso en '0' y el final en '35'(length + espacios en blanco al principio y al final). La condición se obtiene a partir de los paréntesis situados a continuación del 'if'. Una vez extraída se verifica que sea booleana. Más tarde, se obtiene el cuerpo 1 desde donde termina la condición hasta el token 'elif' o 'else'. En este caso, quedaría como condición '3 < 4' y como cuerpo 1 '(1 + 2) * 3 == 9'. Para el cuerpo 2 se toma a partir del token señalado hasta el índice marcado como final(en caso de 'elif' se toman dos índices antes al inicio del cuerpo para tomar el 'if' y retornar una condición). En este caso sería '0':

Finalmente se retorna una tupla de 6 elementos que indican, por orden: si la condicional es válida, condición, cuerpo 1, cuerpo 2, índice de inicio, índice final, expresión global con algunas inserciones de espacios en blanco. Quedaría '(true, 3 < 4, (1 + 2) * 3 == 9, 0, 0, 35)'.

Al llegar estos valores al método de evaluación nuevamente, se evalúa la condición en `Main.Parse` que devolverá 'true'. Por tanto, el cuerpo que se retornará será el 1. Finalmente, se retorna la expresión global con el cuerpo de retorno que corresponde, en el lugar que ocupaba la condicional, quedando '(1 + 2) * 3 == 9'. Y esto ya se ha resuelto anteriormente.

- **Errores:**

Existen 4 tipos de errores: léxico, sintántico, semántico y error en ejecución(overflow). Una vez detectado un error, se llama al método `SetErrors` de `Check`. Aquí se activa el color rojo en la escritura de la consola y se imprime el error directamente. Una vez impreso el error, se activa el campo *error* de `Main`, que evitará que en el retorno de cada llamado recursivo de las funciones del programa, se impriman otros errores. Siempre que se detecta un error, se retorna un string vacío.

El error léxico es aquel que se genera por el uso de tokens inválidos en la entrada:

El error sintáctico es aquel que se genera por una mal escritura de la entrada, o falta de tokens necesarios:

El error semántico es aquel que se genera por el mal empleo de tipos y operadores:

El error en ejecución es aquel que se genera cuando una función recursiva se desborda: