Forge Zero Finders

F0F

Una guía al lenguaje.

Autores:

Niley González Ferrales ..... NileyGF
Arian Pazo Valido ..... ArPaVa

Repositorio del lenguaje:

github.com/NileyGF/F0F

Detalles del lenguaje

FOF es un lenguaje de dominio específico, diseñado para permitir y facilitar la creación de código que encuentre ceros de funciones. Debido a esto, un programa en FOF se compone de una lista de declaraciones auxiliares y finaliza con una función especial Forge, la cual recibe como parámetros la función a la que se le debe encontrar el cero, y los extremos de un intervalo donde se supone que haya al menos un cero.

El lenguaje tiene tipado dinámico y podría ser considerado un lenguaje funcional. Tiene una gramática relativamente simple, y similar a los lenguajes de programación más populares para facilitar la adaptación de nuevos usuarios.

Gramática

Un programa en FOF, como fue expresado anteriormente, se compone de una lista de declaraciones y un Forge. Cada declaración en dicha lista puede ser:

- Una declaración de una función: Consiste en la palabra clave fun, el nombre de la función, entre paréntesis sus parámetros (separados por coma) y entre llaves el cuerpo de la función.

```
fun nombre ( a, b) {
    // cuerpo
}
```

- Una declaración de una variable: Consiste en la palabra clave var, el nombre de la variable y una asignación o un punto y coma.

```
var a = 0;
var b = a;
var a1;
```

- O un enunciado (statement).

Los comentarios son de una sola línea y se escriben después del símbolo '//'.
Destacar también que no se permite más de una variable o función con el mismo nombre, en el mismo ámbito (scope).
En el cuerpo de la función pueden aparecer cualquiera de las declaraciones enunciadas, excepto la de una función. O sea, declarar variables o enunciados.

Ahora listamos los distintos tipos de enunciados (statements).

- while : Este enunciado está compuesto por la palabra clave `while`, (, una expresión condicional, que al evaluarla sea un valor booleano,) y entre llaves: el cuerpo, que acepta lo mismo que el de una función (declarar variables o enunciados).

```
while (condición)
{
    // cuerpo
}
```

- for : Este enunciado está compuesto por la palabra clave `for`, (, la declaración de una variable, con o sin valor asignado, sin olvidar el punto y coma. Luego una expresión condicional, que al evaluarla sea un valor booleano, un punto y coma y una expresión que usualmente es un aumento o decremento de la variable declarada al inicio, seguida de ')'. Por último, entre llaves, el cuerpo del for, que acepta lo mismo que el de una función o un while.

```
for ( declaración de  variable  condición ; expresión )
{
    // cuerpo
}
```

- if : Este enunciado está compuesto por la palabra clave `if`, una expresión condicional entre paréntesis, el cuerpo entre llaves y opcionalmente la palabra clave `else`, con otro cuerpo entre llaves. Ambos cuerpos se componen igual que los explicados en los enunciados anteriores.

```
if (condición) {
    //cuerpo
}
else {
    // cuerpo
}
```

```
if (condición) {
    //cuerpo
}
```

- return : Es un enunciado que solo debe aparecer en algún lugar del cuerpo de una función. Su aparición provoca que se detenga la ejecución de la función que lo contiene. Está compuesto por la palabra clave `return` seguida de una expresión (en ese caso se retorna el resultado de evaluar la expresión) y punto y coma, o solo punto y coma (en este caso no se retorna nada, o sea, null).

```
return 0;
return a + b;
return ;
```

- asignación ; : Este enunciado se compone de un identificador, o sea un nombre de una variable previamente declarada, un símbolo de igual y una expresión (cuyo valor será asignado a la variable). Después de una asignación, **NO OLVIDE EL ;** .

```
a = b + a ;
```

- expresión ; : Este enunciado se compone de una expresión seguida de un punto y coma. (No importa cuantas veces aparezca esta alerta en este tutorial, va a pasar; pero **NO OLVIDE EL ;**)

Como deben haber observado, hay varios enunciados que utilizan una 'expresión', este es el componente más restrictivo de lo visto hasta ahora de la gramática de FOF, ya que disimiles declaraciones lo contienen, pero en una expresión no hay nada de lo visto hasta ahora.

Una expresión en FOF puede ser una operación lógica (`&&, ||`), una comparación (`==, !=, <, <=, >, >=`), operaciones aritméticas (`+,-,*,/,%,^`), operaciones unarias (`!, -`), una invocación de una función, un literal, un identificador o una combinación de cualquiera de las anteriores, parentizando adecuadamente.
A continuación explicaremos que función tienen los símbolos anteriores así como la precedencia y asociatividad que poseen.

- El operador &&, realiza la operación lógica AND entre las expresiones a su izquierda y derecha. Ambas deben ser expresiones booleanas. El operador `||` funciona exactamente igual, excepto porque realiza la operación OR.
- Los operadores de comparación son:
  - `==` : resulta verdadero si las expresiones a comparar tienen igual valor.
  - `!=` : resulta verdadero si las expresiones a comparar tienen distinto valor.
  - `<` : resulta verdadero si el valor de la expresión a la izquierda es menor que el de la derecha.
  - `<=` : resulta verdadero si el valor de la expresión a la izquierda es menor o igual que el de la derecha.
  - `>` : resulta verdadero si el valor de la expresión a la izquierda es mayor que el de la derecha.
  - `<=` : resulta verdadero si el valor de la expresión a la izquierda es mayor o igual que el de la derecha.
- Los operadores aritméticos binarios son; los triviales : `+` para la suma, `-` para la sustracción, `*` para la multiplicación y `/` para la división; `%`

devuelve el resto de la división y ^ efectúa la potencia. En todos los casos, las expresiones a operar tienen que tener un valor numérico.

- Cuando el símbolo - se utiliza al inicio de la expresión se interpreta como el operador unario de negación, la expresión a su derecha tiene tener un valor numérico.
- El símbolo ! también se utiliza como operador unario, en este caso, realiza la operación booleana de NOT, y la expresión a evaluar tiene que ser booleana.

En cuanto a la precedencia, las operaciones con operadores de mayor prioridad se evalúan primero.

Los operadores aparecen en la tabla en orden de prioridad de mayor a menor. Si aparecen varios operadores en la misma fila, es porque tienen igual precedencia.

Operadores	Operación	Asociatividad
`fun(param)`	llamada	Izquierda a derecha
`!, -`	unaria	Derecha a izquierda
`^`	potencia	Derecha a izquierda
`*, /, %`	multiplicativa	Izquierda a derecha
`+, -`	aditiva	Izquierda a derecha
`<, <=, >, >=`	comparación	Izquierda a derecha
`==, !=`	igualdad	Izquierda a derecha
`&&`	AND lógico	Izquierda a derecha
`||`	OR lógico	Izquierda a derecha
`=`	asignación	No asocia

Cuando varios operadores del mismo tipo aparecen en una expresión, la evaluación continúa según la asociatividad. Por supuesto, las operaciones entre paréntesis tienen la mayor prioridad.

Ejemplos:

Expresión	*Parentización implícita*
a && b || c	(a && b) || c
a = b || c	a = (b || c)
1 - 1 - 1 == - 1	((1 - 1) - 1) == - 1
! ! a	! (! a)
2 * 3 ^ 2 ^ 3 == 13122	(2 * (3 ^ (2 ^ 3))) == 13122

Además de una operación, una expresión puede ser un literal como true, false, 88, 3.5, “Hello World!”, o un identificador de una variable o función.

Ejemplos de uso

Ahora procedemos a ejemplificar el uso de la gramática antes expuesta. Para ello pondremos ejemplos de programas en FOF y el resultado.

```
// Forge your best weapon to defeat those functions

fun getmin(a,b)
{
    if (a <= b)
    {
        return a;
    }
    else{
        return b;
    }
}

Forge (f, min, max)
{
    var m = getmin(min,max) ;
    return m ;
}
```

Con `min = -1000` y `max = 1000`. el resultado de `Forge` es:

```
PS C:\Users\Akeso\Documents\VSCode\F0F> py -m F0Fmain Code2.txt
-1000
```

Ahora veamos un ejemplo de una recursividad clásica, cuyo resultado esperado es 55:

```
// Forge your best weapon to defeat those functions

fun fib(n)
{
    if (n <= 0){
        return 0
    }
    if (n <= 2){
        return 1;
    }
    return fib(n-1) + fib(n-2) ;
}
Forge (f, min, max)
{
    return fib(10);
}
```

```
PS C:\Users\Akeso\Documents\VSCode\F0F> py -m F0Fmain Code4.txt
[Parsing error at: } [line 7] : Unexpected token after 0]
```

Este error significa que el lenguaje no espera el token } después del 0. Y es que: OLVIDAMOS EL **;**

```
1   // Forge your best weapon to defeat those functions
2
3   fun fib(n)
4   {
5       if (n <= 0){
6           return 0;
7       }
8       if (n <= 2){
9           return 1;
10      }
11      return fib(n-1) + fib(n-2) ;
12  }
13  Forge (f, min, max)
14  {
15      return fib(10);
16  }
PS C:\Users\Akeso\Documents\VSCode\F0F> py -m F0Fmain Code4.txt
55
```

Podemos subir el nivel ahora y crear un código que en realidad encuentre ceros de funciones.
Para comenzar veremos un ejemplo de código que NO hacer. Porque es tortuosamente lento, y peor si los diseñadores son caprichosos y ponen `min` y `max` a gran distancia (5 unidades de distancia ya es una gran distancia, para este método).

```
1   // Forge your best weapon to defeat those functions
2
3   fun Naive(f, min, max, interval, tolerance)
4   {
5       x = min;
6       while ( abs(x-max) >= tolerance)
7       {
8           if (abs(f(x)) < tolerance){
9               return x;
10          }
11          x = x + interval ;
12      }
13      return x;
14  }
15  Forge (f, min, max)
16  {
17      return Naive(f, min, max, 0.0001, 0.000001);
18  }
PS C:\Users\Akeso\Documents\VSCode\F0F> py -m F0Fmain Naive.txt
Runtime error at: [line 5] : Undefined identifier 'x'.
```

Este nuevo error indica que se está utilizando un identificador de una variable o una función que no ha sido definida. Aquí el problema es que olvidamos el `var`.

```
// Forge your best weapon to defeat those functions

fun Naive(f, min, max, interval, tolerance)
{
    var x = min;
    while ( abs(x-max) >= tolerance)
    {
        if (abs(f(x)) < tolerance){
            return x;
        }
        x = x + interval ;
    }
    return x;
}
Forge (f, min, max)
{
    return Naive(f, min, max, 0.0001, 0.000001);
}
```

Listo. Ahora no hay errores. Pero aún así es una pésima idea ejecutarlo. Así que diseñemos un mejor método de encontrar esos ceros.

Bisección:

```
fun Bisection(f, a , b, repetitions, tolerance)
{
    var fa = f(a);
    var fb = f(b);
    if (fa * fb < 0){
        var i = 0 ;
        while(i < repetitions){
            var c = (a + b) / 2 ; // c = a + (b - a) / 2 ;
            var fc = f(c);
            if (abs(fc) < tolerance || abs(a-b) < tolerance)
            {
                return c ;
            }
            if (fa * fc < 0){
                b = c;
            }
            else{ a = c;}
            i = i + 1 ;
            var fa = f(a);
            var fb = f(b);
        }
        return (a + b) / 2 ;
    }
    return 0;
}
Forge (f, min, max)
{
    if (f(min) * f(max) < 0){
        return Bisection(f, min, max, 40, 0.000001);
    }
    return 0;
}
```

Aunque tiene una precondición, este es un método mucho mejor.
Trucos que pueden aprender de aquí:

- No evaluar en la función más de lo necesario
- Poner un límite máximo de iteraciones
- Nunca utilizar `==` para comparar valores numéricos. Hay quienes consideran esto una maldición imperdonable

Además, amablemente, nosotros predefinimos las funciones `abs(x)` y `derivative(f,x)`.
La primera calcula el módulo de x. Mientras que la segunda calcula, utilizando métodos numéricos, la derivada de la función f, en el punto x.

```
* abs(x - x0)  < tolerance ;
* var df0 = derivative(f, x0) ;
```