Sexto boletín de prácticas sobre MMPor

IG29: Compiladores e intérpretes

Sesiones novena y décima de prácticas

1. Introducción

El objetivo final de esta práctica es dotar a tu calculadora MMPor de una estructura de control repetitiva; por otra parte, esto te llevará a introducir tus primeras comprobaciones semánticas.

2. Estructura de control repetitiva

La estructura repetitiva que debes introducir en tu calculadora es un bucle, en principio infinito, que constará de los siguiente elementos:

- Una línea de cabecera con la nueva palabra reservada `loop`.
- La secuencia de sentencias que habrá que ejecutar repetidamente.
- La línea de cierre con la palabra `end`.

Lógicamente, este bucle será un nuevo tipo de sentencia que, sintácticamente, podrá aparecer en los mismos contextos que las demás sentencias del lenguaje. En particular, no debe existir ningún problema en aceptar varios bucles anidados.

Para dotar de utilidad a un bucle como éste, se necesita algún tipo de sentencia que permita finalizar la ejecución del bucle. La sentencia de ruptura de bucle de MMPor será una línea donde, tras la nueva palabra reservada `break` podrá aparecer, opcionalmente, la palabra `if` seguida de una condición. Si aparece el `if`, que la condición se cumpla o no lo haga determinará si el flujo de ejecución del programa debe saltar a la línea posterior al cierre del bucle al que pertenece o, por el contrario, seguir inmediatamente tras la propia sentencia de ruptura; por su parte, un `break` sin `if` supone salir del bucle de forma incondicional, esto es, se considerará equivalente a éste:

```
break if 1
```

Cabe establecer algunas consideraciones sobre qué contextos admiten una sentencia `break` y cuáles no. En particular, una sentencia `break` sólo podrá aparecer en el interior de un bucle, aunque no se restringe a qué profundidad, es decir, podría aparecer, por ejemplo, dentro de un condicional que estuviera dentro de un bucle... o incluso más anidada. Por otra parte, debe señalarse que una sentencia `break` anidada dentro de varios bucles se asociará siempre al más interior de éstos. Finalmente, todo bucle deberá contar con, al menos, una sentencia `break` asociada.

Un ejemplo de programa con estructuras repetitivas podría ser éste:

```
obj:=77
max:=10
ya:=-1
i:=1
loop
    j:=1
    loop
        if i*j==obj then
            print i, " * ", j, " = ", obj
            ya:=+1
            break
        end
        j:=j+1
        break if j>max
    end
    i:=i+1
    break if ya || i>max
end
```

Como puedes suponer, se trata de que diseñes, implementes y pruebes los cambios necesarios para que tu calculadora admita la presencia de las correspondientes estructuras de control repetitivo en su lenguaje de entrada. Algunas de las modificaciones que necesitas llevar a cabo son similares a las que ya habrás realizado en prácticas anteriores, motivo por el cual no se comentan en este enunciado; otras sí requieren que recibas orientación específica al respecto como es el caso, por ejemplo, del tratamiento de las comprobaciones semánticas. En primer lugar, hay que identificar qué aspectos del análisis de la entrada cabe llevar a cabo semánticamente tras la nueva ampliación del lenguaje MMPor:

a. Garantizar que toda sentencia `break` aparezca dentro de un bucle. En realidad, este control cabría realizarlo sintácticamente, es decir, restringir mediante la correspondiente gramática que las sentencias de ruptura sólo puedan aparecer dentro de bucles; sin embargo, es más cómodo considerar `break` como una sentencia más en la gramática y superponer luego un control semántico.

b. Garantizar que todo bucle tenga al menos una sentencia `break` asociada. También parece más sencillo abordar el problema semánticamente que intentar hacerlo mediante modelado sintáctico.

Respecto a cuándo llevar a cabo las comprobaciones semánticas, lo más cómodo suele ser tener en cada nodo del AST un método específico (por ejemplo: `comprueba`) que primero compruebe la corrección de sus hijos (con llamadas a sus métodos `comprueba`) y luego se preocupe de las comprobaciones específicas del nodo. Así, el esquema de traducción puede limitarse a construir un AST adecuado y dejar las comprobaciones semánticas para una fase posterior, pero anterior a la de ejecución. Por ejemplo:

```
as=AnalizadorSintactico(sys.stdin)
as.arb.comprueba()
```

Dentro de los métodos de comprobación, se deberá llamar, si procede, a la función `errores.trata_error` para mostrar mensajes análogos a los siguientes:

```
ERROR en línea 10 del programa: Sentencia break sin bucle

ERROR en línea 25 del programa: Bucle loop sin sentencia break
```

Esto implica que necesitarás guardar en los nodos de sentencias `break` y bucles sus correspondientes líneas de inicio, información que deberás copiar del atributo `nlinea` que Metacomp ofrece en todos los componentes léxicos emitidos.

Si no se te ocurre una idea mejor, considera la posibilidad de gestionar la información sobre bucles y sentencias de ruptura mediante una lista global de valores lógicos, manejada como una pila del siguiente modo:

- Inicialmente, la pila se crea vacía.
- Cada vez que se entra en un bucle, se apila un valor *false*.
- Cada vez que se ve un `break`, se sustituye por *true* el tope de la pila.
- Cada vez que se sale de un bucle, se desapila un valor lógico.

Así, es fácil detectar los dos tipos de errores semánticos:

a. Sentencia `break` sin bucle: cuando se va a gestionar un `break` y la pila está vacía.

b. Bucle `loop` sin sentencia `break`: cuando el valor lógico desapilado al salir de un bucle es *false*.

El problema de crear métodos de ejecución adecuados para tus nuevos nodos es conveniente que lo dejes para el final, cuando tu calculadora ya sea capaz de analizar correctamente la entrada, detectando los errores que proceda, y de mostrar el correspondiente AST en formato verArbol. Llegados a este punto, una posible forma de abordar la ejecución de bucles y rupturas sería haciendo uso del mecanismo de excepciones de Python del siguiente modo:

- El cuerpo del método de ejecución del nodo bucle puede protegerse mediante una estructura `try/except`.
- La ruptura del bucle puede implementarse entonces en el método análogo del nodo correspondiente elevando una excepción específica que sea la única que se capture (sin llevar a cabo ninguna acción adicional) en el `except` del nodo bucle.

Para crear la excepción específica, bastará con definir una clase global al módulo `AST.py`, por ejemplo así:

```
class BreakMMPor(Exception): pass
```

De este modo, la ruptura del bucle se podrá ejecutar entonces como

```
raise BreakMMPor
```

y el cuerpo del método de ejecución del bucle deberá tener una estructura como sigue:

```
try:
    ...
    ...
    ...
except BreakMMPor:
    pass
```

3. Entrega de la práctica

Como resultado de esta práctica, debes entregar en un paquete `entrega07.tgz` los ficheros siguientes:

- `disenyo.txt`.
- `mmpor.mc`, `AST.py`, `memo.py` y `errores.py`.
- Los correspondientes ficheros de prueba, incluyendo los nuevos.