Segundo boletín de prácticas sobre MMPor

IG29: Compiladores e intérpretes

Cuarta sesión de prácticas

(1) Adapta tu diseño de MMPor para modificar el funcionamiento de la calculadora en el siguiente sentido: la entrada consistirá en una asignación por línea (y no, como hasta ahora, en una expresión por línea):

a. Modifica la especificación léxica para dotarla de dos nuevas categorías léxicas:

- Una para los identificadores, asumiendo que éstos serán secuencias de letras. ¿Se necesita un atributo?
- Otra para el símbolo de la asignación, para el que se utilizará la secuencia de dos caracteres `:=` (estamos, por lo tanto, adoptando el criterio de Pascal).

b. Modifica la gramática para indicar que en cada línea, antes de la expresión ⟨E⟩, debe aparecer un identificador seguido del símbolo de la asignación.

c. Para poder representar asignaciones, añade una nueva clase de nodo al diseño de los ASTs, Asigna, con un atributo (el identificador de la variable receptora) y un hijo (la expresión que hay que evaluar para saber qué valor debe tomar la variable). También debes indicar que los hijos de un nodo Secuencia ya no serán expresiones, sino asignaciones.

d. Modifica la gramática para indicar que, donde pueda aparecer un literal numérico, también podrá aparecer un identificador de variable.

e. Añade una nueva clase de nodo al diseño de los ASTs, Vble, para representar apariciones de identificadores de variable en partes derechas de asignaciones. Los nodos de esta clase tendrán un atributo: el identificador de la variable.

f. Modifica el esquema de traducción para reflejar los cambios en la gramática y cómo han de construirse los ASTs:

- A la lista auxiliar *laux* habrá que ir añadiéndole nodos Asigna en vez de, simplemente, los ASTs obtenidos al analizar ⟨E⟩.
- El análisis de un identificador generado por ⟨F⟩ deberá dar lugar a la construcción del correspondiente nodo Vble en ⟨F⟩.*arb*.

(2) Modifica y completa la implementación para adaptarla al nuevo diseño:

a. Traslada las modificaciones de la especificación léxica a las primeras líneas de `mmpor.mc`.

b. Crea un nuevo módulo, `memo.py`, donde se defina como sigue una clase `Memoria` para poder gestionar los valores de las variables:

```
class Memoria:

    def __init__(self):
        self.tabla={}

    def asigna_vble(self,ident,valor):
        self.tabla[ident]=valor

    def existe_vble(self,ident):
        if ident in self.tabla:
            return True
        else:
            return False

    def recupera_vble(self,ident):
        if self.existe_vble(ident):
            return self.tabla[ident]
        else:
            return None
```

Después, crea un objeto `M` de la clase `Memoria` en el módulo `AST.py`; `M` será una variable global en ese módulo y representará la memoria de la calculadora

c. Crea un nuevo módulo, `errores.py`, donde se defina una función `trata_error` con las siguientes características:

- Debe aceptar dos parámetros: `lugar` y `lo_que_pasa`.
- Debe imprimir por la salida de error (`sys.stderr`) una línea con el siguiente formato:

  ERROR en línea *lugar*: *lo_que_pasa*

- Finalmente, debe detener la ejecución del programa con `sys.exit(1)`.

d. Crea en el módulo `AST.py` una nueva clase, `Asigna`, que:

- Acepte en su constructor un atributo identificador y un hijo expresión.
- Tenga un método `ejecuta` para evaluar su hijo, asignar el valor resultante (con el método `asigna_vble` de `M`) a la correspondiente variable receptora y, finalmente, imprimir en una línea de la salida estándar:
  - El identificador de la variable.
  - Un espacio en blanco.
  - El valor asignado.
- Tenga un método `__str__` que devuelva la cadena siguiente:

  `*** Todavía sin implementar ***`

e. Modifica adecuadamente la implementación de la clase Secuencia (por ejemplo, ya no habrá que evaluar e imprimir resultados de expresiones, sino simplemente ejecutar asignaciones).

f. Crea en el módulo `AST.py` otra nueva clase, `Vble`, que:

- Acepte en su constructor el correspondiente atributo.
- Tenga un método de evaluación que consulte si la correspondiente variable existe en `M` y obre en consecuencia:
  - Si existe, el método debe devolver el valor de la variable.
  - Si no, debe llamar a `trata_error` para proporcionar un mensaje de adecuado, indicando que el lugar es `del programa`.
- Tenga un método `__str__` que devuelva la cadena siguiente:

  `*** Todavía sin implementar ***`

g. Ahora ya puedes intentar incorporar al esquema de traducción de `mmpor.mc` las correspondientes modificaciones introducidas en el diseño.

h. Finalmente, incorpora a todos tus módulos Python la línea siguiente para evitar quejas del intérprete por utilizar caracteres acentuados:

```
# -*- coding: latin-1 -*-
```

Debe ser la primera línea en cada fichero.

(3) Utiliza Metacomp para transformar `mmpor.mc` en un programa Python de nombre `mmpor` y pruébalo siguiendo un esquema similar al de la práctica anterior. Cuanto más exhaustivas sean las pruebas, mejor.

(4) Entrega en un paquete `entrega03.tgz` los ficheros siguientes:

- `disenyo.txt`, tal como haya quedado después de la adaptación.
- `mmpor.mc`, `AST.py`, `memo.py` y `errores.py`.
- Los correspondientes ficheros de prueba.

(5) Puedes ir adelantando qué modificaciones serían necesarias para dotar a nuestra calculadora de la siguiente interfaz con el usuario:

- Si el usuario llama a la calculadora sin argumentos, ésta debe funcionar normalmente.
- Si el usuario la llama con una opción `-s`, la calculadora debe, después de construir el AST de la entrada, imprimirlo por la salida estándar (en vez de ejecutarlo). Además, el formato de impresión debe ser tal que permita visualizar el AST mediante el programa verArbol.
- Si el usuario llama a la calculadora con más de un argumento o con uno que no es `-s`, ha de utilizarse `trata_error` para dar un mensaje de error adecuado, indicando que el lugar es `de órdenes`.