Séptimo boletín de prácticas sobre MMPor

IG29: Compiladores e intérpretes

Sesiones undécima y duodécima de prácticas

1. Introducción

El objetivo final de esta práctica será introducir en tu calculadora MMPor un nuevo tipo para representar puntos de coordenadas enteras en un espacio bidimensional; además, habrá que definir explícitamente al comienzo de todo programa MMPor las variables que vayan a ser utilizadas. Esto supondrá tener que extender el nivel semántico de la calculadora para dotarlo de gestión de tipos y tabla de símbolos.

2. Tipo de datos punto

Hasta ahora, MMPor sólo contaba con el tipo entero y, de una forma muy restringida, con el tipo cadena: sólo literales de cadena y sólo en sentencias de escritura. Se trata ahora de añadir un nuevo tipo que pueda intervenir, como el entero, en las expresiones del lenguaje: el punto. Los objetos de este tipo serán puntos de coordenadas enteras en un espacio bidimensional y no tendrán literales en MMPor: en su lugar, se podrá construir un punto a partir de dos expresiones enteras separando las expresiones mediante una coma y encerrando el par entre paréntesis. Otras expresiones relacionadas con el tipo punto, en las que intervienen operandos de este tipo, utilizan los siguientes operadores:

- Unarios prefijos: `+ -`
- Multiplicación: `*`
- Suma y resta: `+ -`
- Comparadores: `== !=`

Si p es un punto `(p1,p2)`, q es un punto `(q1,q2)` y x es un entero, entonces:

- `+p` devuelve el punto p.
- `-p` devuelve el punto `(-p1,-p2)`.
- `x*p` devuelve el punto `(x*p1,x*p2)`.
- `p*q` devuelve el entero `p1*q1+p2*q2`.
- `p+q` devuelve el punto `(p1+q1,p2+q2)`.
- `p-q` devuelve el punto `(p1-q1,p2-q2)`.
- `p==q` devuelve el entero `p1==q1&&p2==q2`.
- `p!=q` devuelve el entero `p1!=q1||p2!=q2`.

Y, por supuesto, una expresión de tipo punto puede encerrarse entre paréntesis.

De momento, las expresiones de tipo punto sólo podrán emplearse, aparte de para construir otras expresiones (y únicamente de las formas que se han señalado anteriormente), en sentencias de escritura; la forma de mostrar un punto por la salida estandar será escribiendo sucesivamente, y sin ningún blanco intermedio: un paréntesis abierto, el primer entero del punto, una coma, el segundo entero y un paréntesis cerrado. A los puntos no se les asociará valor lógico alguno, así que no podrán ser utilizados como condiciones en estructuras de control. Como estamos suponiendo variables de tipo entero, las expresiones de tipo punto tampoco podrán intervenir, todavía, en asignaciones.

Conviene no intentar evitar sintácticamente errores por un mal uso de los tipos; es muy preferible mantener la gramática simple y añadir las correspondientes comprobaciones en el nivel semántico. Los mensajes de error podrían ser del siguiente estilo:

```
ERROR en línea 15 del programa: Tipos incompatibles en ! POINT
```

```
ERROR en línea 20 del programa: Tipos incompatibles en ( POINT , INT )

ERROR en línea 25 del programa: Tipos incompatibles en POINT * INT

ERROR en línea 30 del programa: Tipos incompatibles en POINT && POINT

ERROR en línea 35 del programa: Condición de tipo POINT

ERROR en línea 40 del programa: Intento de asignar POINT a receptor INT
```

Para poder llevar a cabo las correspondientes comprobaciones, parece razonable dotar de un atributo *tipo* a cada nodo AST que represente alguna clase de expresión. Siguiendo el mismo estilo utilizado hasta ahora, correspondería calcular tipos en los constructores de los nodos y comprobar los posibles errores a posteriori, en los métodos `comprueba`. Aunque es posible proceder así, puede que te resulte más cómodo posponer el cálculo de tipos a los métodos `comprueba` y centralizar allí toda su gestión (cálculo más comprobaciones).

Para introducir en tu calculadora todos los cambios que se te piden en este apartado, puedes seguir, por ejemplo, estos pasos:

(1) Crea un módulo `puntos.py` donde se defina una clase `Punto` con los métodos que luego vayas a necesitar en `AST.py`: constructor, cambio de signo, producto por un escalar, producto escalar, suma, resta, igualdad, desigualdad y representación como cadena.

(2) Modifica convenientemente tu diseño en `disenyo.txt`.

(3) Adecúa tu implementación al nuevo diseño, incorporando también el correspondiente tratamiento de errores y sin olvidar los métodos `__str__` de los nodos del AST.

(4) Crea nuevos ficheros p*XX*`.mmp` y p*XX*`.sal` para probar las nuevas capacidades de tu calculadora. Verifica que tu calculadora supera con éxito las nuevas pruebas y que tampoco falla con las anteriores. Comprueba también el funcionamiento de la opción `-s`.

3. Definición explícita de variables

Hasta ahora, todas las variables eran de tipo entero y podían empezar a utilizarse en cualquier momento; en adelante, será necesaria la definición explícita de toda variable que vaya a aparecer en el programa (independientemente, además, de que luego el flujo de ejecución vaya a alcanzar o no al uso de esa variable). En la definición de una variable se especificará el tipo de ésta, que ahora podrá ser punto, y no sólo entero. Esto propiciará la aparición de un nuevo error de tipos que antes no podía darse: el de intentar asignar el valor de una expresión entera a una variable de tipo punto. Aparte, como veremos más adelante, de los errores semánticos típicamente asociados a la gestión de una tabla de símbolos: el intento de definir una variable con un identificador ya utilizado para tal fin y el de hacer uso en una sentencia de un identificador no aparecido en definición alguna.

El lenguaje MMPor ofrecerá, pues, una nueva construcción sintáctica: las líneas de definición, en cada una de las cuales podrán ser definidas varias variables, todas ellas del mismo tipo. Concretamente, una línea de definición comenzará con la especificación del tipo de las variables mediante una de las dos nuevas palabras reservadas: `int` (entero) y `point` (punto); además, entre el nombre del tipo y el correspondiente salto de línea, deberá aparecer una lista no vacía de identificadores (los de las variables que quedarán definidas con el tipo especificado) separados por comas. Las líneas de definición, de haberlas, serán siempre las primeras del programa, esto es, tras la primera sentencia (una línea de definición no lo es) ya no podrá definirse ninguna nueva variable.

La definición de variables obliga a gestionar nueva información asociada a sus identificadores, mediante el objeto global que denominaremos *tabla de símbolos*. Es importante distinguir entre la tabla de símbolos y el objeto de la clase `Memoria` que se utiliza para guardar los valores de las variables:

- La tabla de símbolos se utilizará para almacenar la información sobre las variables que sea pertinente conocer durante el análisis de la entrada.
- La memoria de la calculadora se utiliza para gestionar los valores que las variables van tomando en tiempo de ejecución.

No obstante, la definición de la clase `TablaDeSimbolos` puede ser muy similar a la de `Memoria`, con métodos `define_v` (para definir una variable con su tipo asociado), `existe_v` (para saber si una variable ya ha sido definida) y `recupera_v` (para averiguar el tipo de una variable).

Observa que, como ya se comentó, la definición explícita de variables expone tus programas a dos nuevos tipos de errores semánticos: el intento de definir una variable con un identificador ya utilizado en una definición previa y el de utilizar en una sentencia un identificador con el que no se ha definido variable alguna. Por ejemplo:

```
ERROR en línea 15 del programa: Identificador "x" ya utilizado en definición

ERROR en línea 20 del programa: Uso de identificador "j" no definido
```

Cabe insistir en el diferente tratamiento que merecen las variables en análisis (gestión de tipos y posibles errores semánticos) y en ejecución (gestión de valores y posibles errores de ejecución). En particular, intentar utilizar como variable un identificador que no haya intervenido en una definición previa (error semántico, detectado en las correspondientes comprobaciones de la fase de análisis) será muy distinto de intentar acceder al valor de una variable definida pero que no haya sido previamente inicializada (error de ejecución, detectado en el método de evaluación del correspondiente nodo identificador).

Como puedes suponer, se trata de que tengas en cuenta todo lo anterior para diseñar, implementar y probar los cambios necesarios conducentes a que tu calculadora pase a exigir la correcta definición, explícita, de toda variable que luego vaya a ser utilizada en las sentencias del programa.

4. Entrega de la práctica

Como resultado de esta práctica, debes entregar en un paquete `entrega08.tgz` los ficheros siguientes:

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