[This translation of an earler version of the first part of the talk is graciously and gratefully provided by Gonzalo Peña-Castellanos]

Hola, soy Rocky, un desarrollador de código abierto de vieja data. Paso la mayor parte del tiempo detrás de mi computadora escribiendo código en lugar de dar charlas. Pero ocasionalmente, tengo una muy buena idea que quiero compartir. Entonces aquí estoy!

Como dije, paso mucho tiempo escribiendo código, pero eso también significa que paso mucho tiempo depurando código, porque cometo un montón de errores. Incluso cometo errores al encontrar mis errores. Y porque yo cometo tantos errores, he escrito tal vez una docena de depuradores. Los que he escrito para Python creo que son bastante únicos, y haré una demostración de uno de ellos, pero primero hablaré de un problema en particular.

¿Cómo podemos determinar con precisión dónde se encuentra un programa cuando estamos ejecutando un depurador o cuando ocurre un error?

A veces recibimos mensajes de error en Python como este:

```
Traceback (most recent call last):
  File "bug.py", line 2, in <module>
    i / j / k
ZeroDivisionError: integer division or modulo by zero
```

El mensaje da el nombre de archivo "bug.py" y el número de línea del error, línea dos. Incluso da el texto de la línea dos. ¡Esto es genial! Pero hay *dos* divisiones en la línea dos. ¿De cuál estamos hablando?

Aquí hay otros ejemplos:

```
x = prev[prev[0]]                       # which prev ?
[e[0] for i in d[j] if got[i] == e[i]] # lots going on here
exec(some_code % 10, namespace)         # code created at runtime
```

Si obtenemos un error al suscribir "prev", de cuál acceso "prev" estamos hablando?

En este segundo ejemplo, hay varios accesos a arreglos diferentes, e incluso si conocemos el valor del índice (0 o lo que sea), seguimos sin saber a cual lista se accede. Mostraré cómo podemos desambiguar estos.

El último ejemplo, creo, es particularmente interesante porque muestra que en Python podemos ejecutar código que se crea en tiempo de ejecución y el el código fuente no se encuentra en un archivo en alguna parte.

Aquí hay un mensaje de error que podemos obtener al ejecutar eso:

```
Traceback (most recent call last):
  File "bug-exec.py", line 3, in <module>
    exec(template % 10, namespace)
  File "<string>", line 1, in <module>;
NameError: name 'bad' is not defined
```

¡Problema! File ""? Y qué variable, x o y, fue la que contenía "bar"?

Veamos como podemos hacerlo mejor. Pero para mostrar como, tenemos que desviarnos un poco para comprender cómo la computadora "ve" el programa en comparación con como lo vemos nosotros

En la implementación de C Python, el código que escribimos se traduce en algo llamado bytecode. Entonces un programador puede escribir:

```
x + y
```

pero después de la compilación el bytecode que ve la computadora es:

```
  2           8 LOAD_NAME                0           x
             10 LOAD_NAME                1           y
             12 BINARY_ADD               None
```

El "2" en la primera línea significa que las siguientes instrucciones vienen de la línea dos, hasta que se dé otro número de línea. Aquí, no hay otra línea con números, por lo que el intérprete de Python asume que todos las instrucciones tienen algo que ver con la línea 2.

Los números 8, 10 y 12 son las direcciones/desplazamientos de las instrucciones, o realmente instrucciones diferenciales desde donde comienza la función o módulo. Asi que siendo más preciso sobre lo que está asociado con la línea 2, la respuesta es: instrucciones en los desplazamientos 8, 10 y 12. En las versiones de Python anteriores a 3.6, la instrucción para `LOAD_NAME` es de 3 bytes en lugar de 2, por lo las lineas serían 8, 11 y 14.

Y esto (o una versión más oscura codificada en formato binario) es lo que la computadora "ve" y ejecuta.

En este simple ejemplo, parece un solo token en el el código fuente se convierte a una sola línea en el bytecode. sin embargo el orden es un poco diferente: el "agregar" viene al final en vez del medio.

Compara eso con:

```
Spanish:
mango fragante

English:
```

```
fragrant mango
```

Esto es bastante simple: simplemente cambiamos el orden de las palabras y hacemos algunos pequeños cambios de ortografía. Esto sucede en la traducción a bytecode también.

Ahora considera:

```
Spanish:
Entiendo

English:
I understand
```

Aquí, una palabra en español se convierte en dos palabras en inglés. Esto también ocurre en la traducción de Python a bytecode.

Ahora considera:

```
Spanish:
templado

English:
not hot and not cold
```

Qué parte de "templado" es "not hot" y qué parte es "and not cold"? El hecho es que no hay una palabra simple y equivalente en inglés.

Este tipo de cosas *nunca* suceden en la traducción de Python a bytecode. E intencionalmente. Afortunadamente para nosotros, el bytecode y el código fuente fueron diseñados sobre los mismos tipos de conceptos.

Entonces, ¿cómo le damos sentido a esto? Déjame sugerir una respuesta señalando que probablemente hayas visto ejemplos de este tipo de situaciones cuando lees una traducción comparando los dos textos lado a lado:

```
English:
Whose woods these are I think I know.
His house is in the village though;

Spanish:
A quién pertenece este bosque creo que sé
Aunque su casa en el pueblo está
```

Puedes suponer que la primera línea del inglés es más o menos como la primera línea del español, y la segunda línea del inglés es como la segunda línea del español. Y posiblemente las palabras en el principio o final de la oración aproximadamente ir con palabras en el comienzo de la traducción

correspondiente. "His house" en el el comienzo de la línea dos es casi la misma posición que "su casa". Pero no exactamente, y algunas veces la posición no concuerda en absoluto. por ejemplo, vea las posiciones de "is" y "though" contra "Aunque" y "está" en la línea 2.

Y esto se parece a lo que sucede cuando recibimos el mensaje de error que vimos arriba, cuando dijo que el error estaba en la línea 2.

```
Python:
x = 1
y = 2

Bytecode:
  1:           0 LOAD_CONST               0 (1)
               2 STORE_NAME               0 (x)

  2:           4 LOAD_CONST               1 (2)
               6 STORE_NAME               1 (y)
```

Observa de nuevo como el orden de los nombres de las variables y de las constantes se ha invertido como estaban en el ejemplo con la adición. El nombre (matemático) para este tipo de orden es "inversa Polaca". Los adjetivos en inglés son así.

Ahora que hemos visto cómo esto se parece al problema de traducción de lenguaje humano, en lugar de un problema de programador versus computadora, podemos preguntar: ¿cómo obtendríamos más detalles en las correspondencias entre las traducciones?

Estoy seguro de que muchos de ustedes ya conocen la respuesta: desglosar la oración en sus partes gramaticales. Asi que

[Mostrar gramática para "Yo como mondogo" y "Como mondongo"]

Bajo los conceptos principales de la gramática, son lo mismo: una oración está compuesto por una sujeto y un predicado. En español, el sujeto puede no estar mientras que en inglés es obligatorio. He simplificado el ejemplo eliminando conceptos gramaticales como "frase nominal", "frase verbal", etc. Añadir estos no cambia sustancialmente las cosas para hacer la diapositiva más difícil de leer

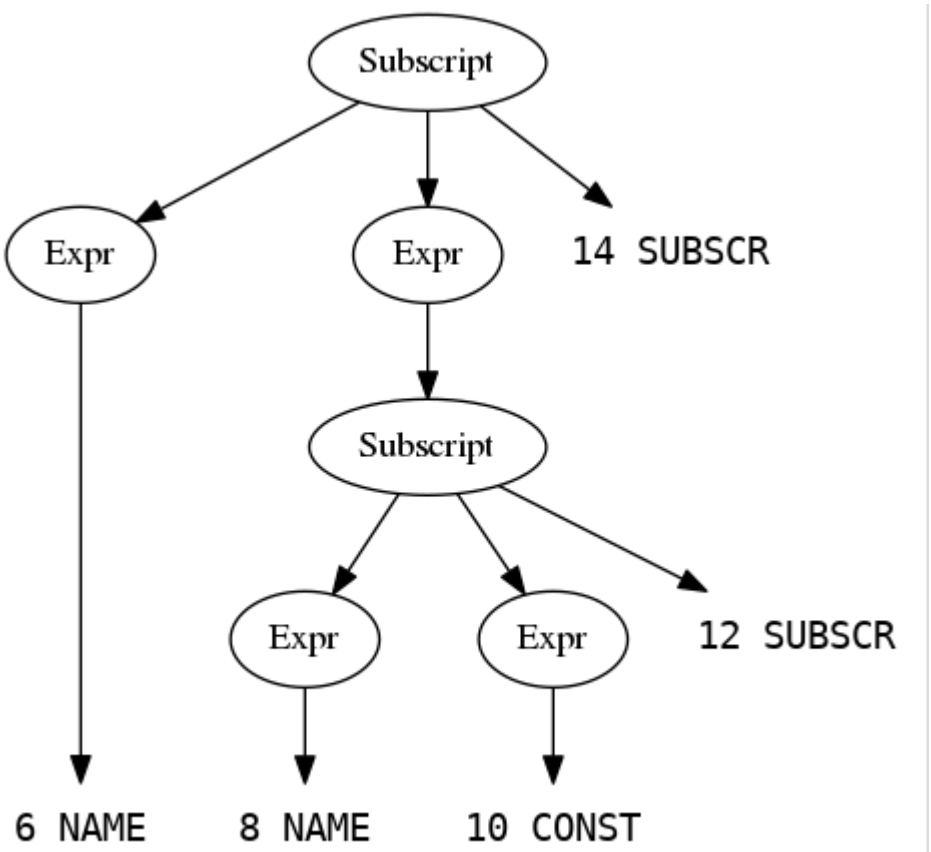
Ahora veamos cómo esto se aplica a la traducción de Python a bytecode ...

```
Python:
prev[prev[0]]
```

se traduce a:

```
2   6 LOAD_NAME "prev"
    8 LOAD_NAME "prev"
   10 LOAD_CONST 0
```

```
  12 BINARY_SUBSCR
  14 BINARY_SUBSCR
```



analiza a:

```
binary_subscr
  0. expr
     L.   2       6  LOAD_NAME 'prev'
  1. expr
     binary_subscr
     0. expr
                  8  LOAD_NAME 'prev'
     1. expr
                 10  LOAD_CONST 0
     2.          12  BINARY_SUBSCR
  2.          14  BINARY_SUBSCR
```

Observa como todas las hojas del árbol de análisis son las instrucciones en bytecode. Puede ser difícil de ver porque las hojas están en diferentes profundidades de los árboles. Pero como verán, esto se volverá importante un poco más tarde.

La representación ASCII no gráfica es más compacta que el diagrama incluso si el diagrama es más bonito.

Ahora estamos en condiciones de mostrar las correspondencias entre los desplazamientoy dónde estamos en el programa

Si estamos en el desplazamiento 12, ejecutamos:

```
prev[prev[0]]
     -------

Contained in...
Grammar Symbol: subscript
prev[prev[0]]
-------------
```

El padre de la instrucción, `subscript`, entiende cómo crear fragmentos de texto de Python. En particular, sabe que debe agregar `[]` alrededor de su segundo hijo, y que el tercer hijo no contribuye texto - está justo allí.

Si, por otro lado, estamos en el desplazamiento 14, ejecutamos:

```
prev[prev[0]]
-------------
Contained in...
Grammar Symbol: assign
x = prev[prev[0]]
-----------------
```

Tiempo de una demostración: demo

Hay una gran cantidad de tecnologías e ingeniería que hace que todo esto funcione. De hecho, hay muchos errores, y al dar esta charla me he obligado a corregir algunos de ellos, al menos los que necesitaba corregir para hacer las demostraciones en vivo.

Ya he hecho esto para Python y para Perl. Sin embargo, la idea es bastante general y se puede aplicar a otros idiomas. La wiki para el descompilador de Python https://github.com/rocky/python-uncompyle6/wiki tiene mucha información sobre cómo funciona el código, e incluso hay un documento con una versión más técnica y orientada a la investigación para quienes les gustaría llevar esta idea a otros idiomas. Los animo a hacer esto.

Gracias.

Links

- Python Decompiler
- Python Bytecode Library
- traceback module + deparsing
- Python 3 Debugger
- trepan2 Documentation
- Emacs Interface to Debuggers
- Decompilation Research Paper

- This presentation