Diseño de Compiladores 2015

Fecha: 16/07/15

Grupo:

Ignacio Betancurt - 4.618.336-3

Alejandro Brusco - 4.309.187-4

Andrés Vera - 4.547.166-8

Nombre del docente:

Pablo Garbusi

1. *Índice de Contenidos*

Índice de Contenidos

Resumen

Aspectos de la Gramática

Gestión de Errores

Representación Intermedia

Stack

¿Cómo ejecutar?

Casos de prueba

Errores conocidos

2. Resumen

El presente documento pretende describir los aspectos técnicos utilizados para la implementación de intérprete de Python solicitado. En el mismo describiremos a grandes rasgos la gramática definida, así como también la gestión de los distintos tipos de errores manejados y la representación intermedia utilizada. Por último se describirán los casos probados y que intenta contemplar cada uno.

3. Aspectos de la Gramática

La definición de la gramática parte de un símbolo inicial "program" (no terminal). Del mismo se desprende un encadenado de sentencias. Dichas sentencias pueden ser "simples" o "complejas".

Sentencias simples: en este conjunto entran las sentencias: "print", "asignación", "return", "continue", "expresión". Esta última sentencia hace referencia a operaciones (con y sin variables) que no son asignadas. Cabe destacar que la separación de las sentencias anteriores, por un separador de "," y en una misma línea, entran dentro de este tipo de sentencias.

Sentencias compuestas: en este otro conjunto entran el resto de las sentencias, que como dice su nombre son más complejas. Estas son: "if", "if-else", "while", "for", "definición de función".

Para el caso de la sentencias simples, cada una de ellas se compone por "expresiones". Las expresiones pueden ser cualquier tipo primitivo de Python (int, long, float, list, dict, tuple, string) y la combinación de operaciones posibles entre ellos. Como

expresión, también se tiene la llamada a una función, ya que esta puede devolver un valor a utilizar en una operación y siguiendo la misma lógica la sentencia “raw_input” es una función que devuelve el String leido, por lo tanto es también del conjunto de expresiones.

Por último, las sentencias complejas, tienen siempre asociado una “suite” de sentencias que representan el cuerpo de lo que se desea ejecutar en el contexto de la sentencia, por ejemplo sentencias dentro de una sentencia “if”.

4. Gestión de Errores

Para el chequeo de posibles errores se tienen varios niveles bien definidos. Los mismo son los siguientes:

- Errores en lectura de tokens
- Errores en Generando el árbol sintáctico
- Errores semánticos en la evaluación del árbol
- Errores en tipos
- Errores en variables/funciones no definidas

4.1. Error en lectura tokens

Para la etapa del análisis léxico y la lectura de tokens, se definió un mensaje predeterminado, que indica que se está leyendo un carácter que no se espera y no puede reconocerse. El mismo indica la línea y la columna en la que ocurrió la lectura no esperada. El mensaje tiene la siguiente forma:

"Illegal character at line 9, column 9"

4.2. *Errores generando el árbol sintáctico*

Para la etapa del análisis sintáctico, en caso de que el parser no pueda reducir o shiftear, por no encontrar una producción correspondiente, esto quiere decir que tenemos un error en la sintaxis de nuestro programa. Para esto, se muestra un mensaje similar al punto anterior indicando la línea donde se encuentra el error:

"Syntax error at line 9, column 9"

4.3. *Errores semánticos en la evaluación del árbol*

En dicha etapa, se debe chequear que las sentencias se encuentran en su contexto, como por ejemplo las sentencias "continue", "break" o "return". En este caso, también se muestra la línea en la cual se encuentra el error y una descripción breve del mismo. El error mostrado puede ser como los siguientes:

"Error at line 6: 'return' sentence not in Function Definition"
"Error at line 6: 'break' sentence not in Iteration Definition"
"Error at line 6: continue sentence not in Iteration Definition"

4.4. *Errores en tipos*

Dichos errores, forman parte del análisis sintáctico. Cuando detectamos un error de tipos en una expresión aritmética, lógica, condición de sentencia

(if,if-else, while, cantidad de argumentos de una función) lanzamos este tipo de error. El error mostrado puede ser como los siguientes:

"Error at line 10: function 'sumar' must be called with 3 arguments"
"Error at line 8: operation 'int + str' is not defined."

4.5. Errores en Variables/Funciones no definidas

Este tipo de errores, son mostrados cuando en tiempo de ejecutar, una variable o función con cierto identificador no se encuentra definida. Los mensajes son de la siguiente manera:

"Error at line 4: function 'factorial' is not defined"
"Error at line 7: variable 'variable1' is not defined"

5. Representación Intermedia

La representación intermedia, se encuentra representada por distintas clases java, siguiendo un modelo de herencias, para poder así construir el árbol sintáctico. La clase principal se llama "Program". Por otro lado, cada sentencia se representa con la clase abstracta "Statement" y cada expresión se representa por la clase "Expression". Por lo tanto, la clase "Program" tiene como atributo una lista de "Statement". Cada clase de estas, tiene siempre un método "eval", que "ejecuta" lo que corresponde (el programa, la sentencia o la expresión). La idea consiste en llamar únicamente al "eval" de "Program" y que este, comience a encadenar en la ejecución las llamadas a "eval" de las diferentes sentencias y luego estas a las expresiones. De esta manera modelamos el árbol.

Puesto que no todas las sentencias son iguales (tenemos if, while, etc) y que no todas las expresiones son iguales (podemos tener llamada a función, sumas, restas, etc) tenemos para cada tipo de sentencia/expresión una clase que la modela y extiende de "Expresion" (lo que lo obliga a definir su "eval" concreto). Es en esta clase, donde el código del "eval" refleja el comportamiento de la Expresión. La imágen a continuación muestra a grandes rasgos el esquema de clases que se definieron.

Diagrama de Estructura de "Statement"

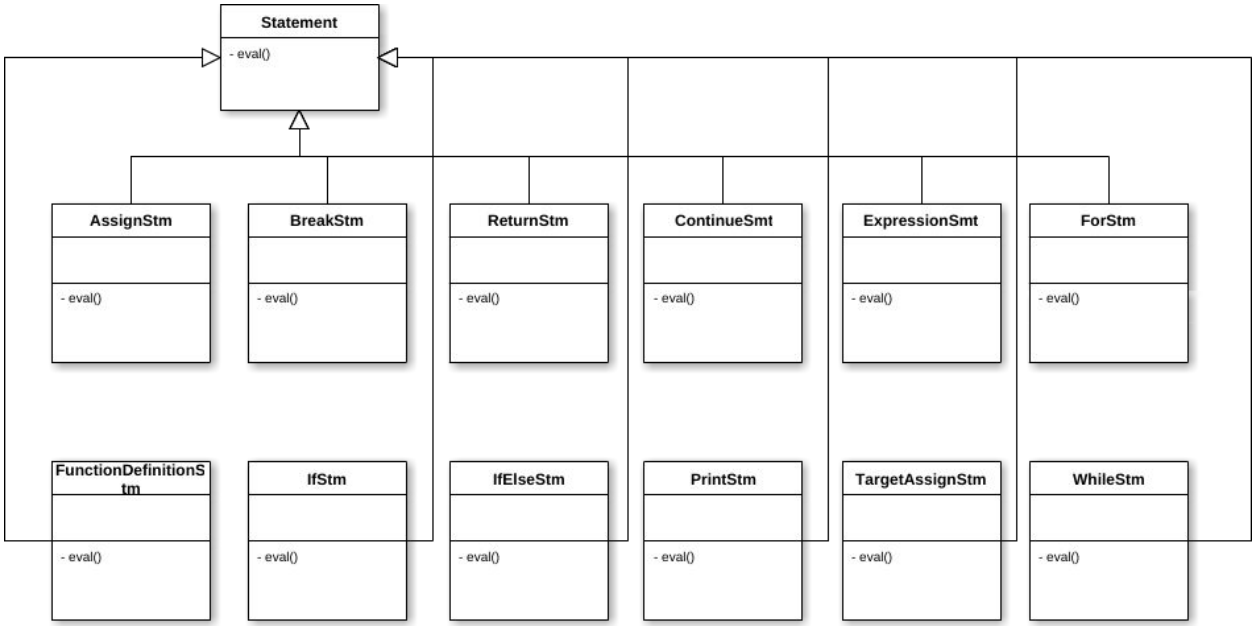
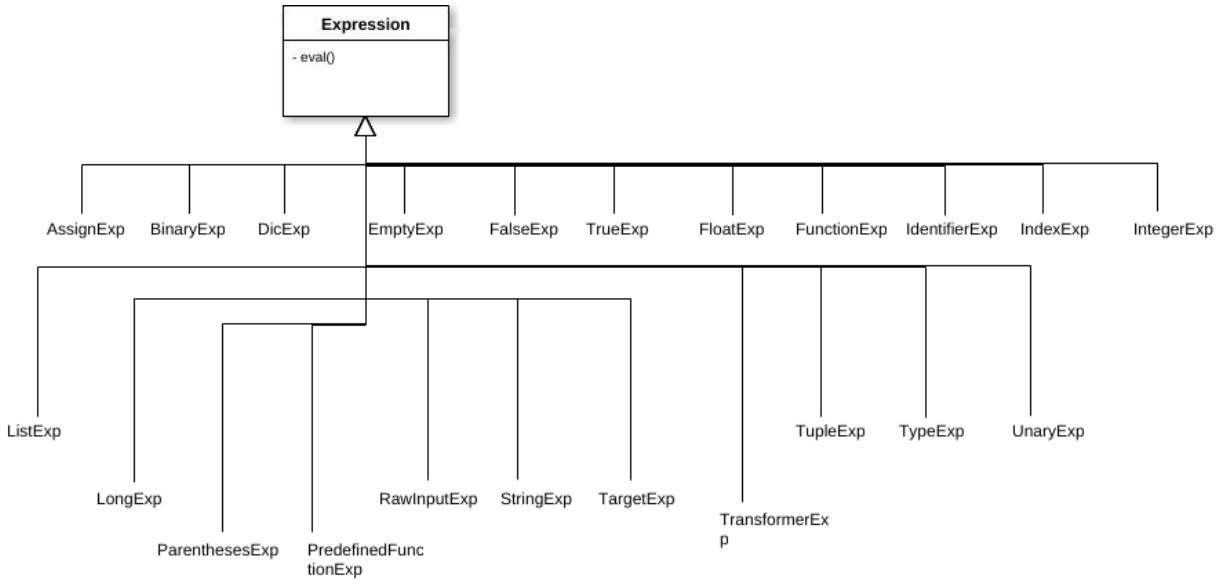


Diagrama de Estructura de “Expression”



Observaciones:

- Llamamos “TargetExp” a expresiones donde se accede a índices como a[1].
- Usamos la expresión “IndexExpr” para describir la posible sintaxis de acceso a listas como lo puede ser “1:2:3” en la expresión “a[1:2:3]”
- La expresión “TransformerExp” hace referencia a expresiones del tipo “int(X)”, “float(X)” y las restantes.

Cabe mencionar, que el método “eval” para las expresiones retorna un objeto de tipo “Types” que es también abstracto, ya que representa la evaluación de una expresión, y esta puede ser una lista, un entero, un string, etc. La clase “Types” nos brinda los métodos necesarios para quien la utilice, conozca qué tipo de dato es el que almacena, para luego poder acceder a sus valores.

6. *Stack*

El Stack diseñado para la oportunidad, refleja distintos aspectos de un scope. Esto incluye, variables en un scope dado, funciones definidas en un scope dado o banderas de “se puede return”, “se puede break”, “se puede continue” en el scope dado.

Primero que nada se tiene un objeto “StackHandler” que tiene el manejo de los distintos scopes. Esta clase, es singleton, ya que nos interesa que se tenga el mismo Stack en todos los momentos de la ejecución del intérprete.

Componentes de “StackHandler”:

- *Scope variables Control:*

  Este valor, contiene una lista (que cada elemento representa un scope). Dentro de cada nodo de la lista, se tiene un par de variables booleanas, para saber si en dicho Scope es posible ejecutar “continue” y “break”. Estos valores, se ven modificados en el scope dado, cuando se arranca a ejecutar Iteraciones (sentencias while/for).

- *Scope sentencia Return:*

  Similar al punto anterior, se mantiene una lista que representa el scope de ejecución, donde se tiene un booleano que indica si es posible ejecutar una sentencia “return”. Dicho valor es modificado en el scope cuando se ejecuta una sentencia de invocación a función.

- Scope Funciones:

  Dicha variable, es una lista representando al scope dado, donde cada nodo de la lista tiene un mapa “id”,”sentencias función”. De esta manera, podemos conocer las funciones definidas en un scope dado.

- Scope Variables:

  Dicha variable, es una lista representando al scope dado, donde cada nodo de la lista tiene a su vez otra lista, que representa las variables definidas en el scope.

Cabe destacar que cada vez que se requiera buscar una variable o función dentro de un scope, estas listas se recorren desde el scope actual de la ejecución, hasta el principal, esto es, recorrer las listas mencionadas anteriormente desde fin hasta inicio.

7. *¿Cómo ejecutar?*

7.1. Compilación

Se entrega un archivo "build.xml" (ubicado en language/build.xml) que contiene las directivas necesarias para poder compilar el proyecto a través de la herramienta **Apache ANT**. Situados en el directorio language/, el comando ant se encargará de compilar el léxico, la gramática, el proyecto JAVA, y generará un JAR con el mismo.

Por otro lado, este proyecto requiere que se utilice la versión de JAVA 1.8 o superior. De todos modos, al ejecutar el comando ant se valida el cumplimiento de esta condición y fallará la compilación en caso de no cumplirse.

Ejemplo de ejecución:

C:\Compilador\tests\> ant

7.2. Ejecución

Luego de compilado el proyecto, se podrá ejecutar el mismo a través del script ubicado en la raíz del entregable con el nombre:

"ejecutar_individual_con_arg_linux.sh"(Linux)

"ejecutar_individual_con_arg.bat"(Windows)

Este script requiere que se le pase la ruta de un archivo Python como argumento.

Ejemplo de ejecución (Windows):

C:\Compilador\> ejecutar_individual_con_arg.bat C:\Compilador\test.py

Ejemplo de ejecución (Linux):

username@host:~/Compilador$ ./ejecutar_individual_con_arg_linux.sh ~/test.py

8. *Casos de prueba*

8.1. *Suite de archivos*

Los casos de prueba presentados en el entregable pretenden mostrar y probar cada uno de los aspectos requeridos para el intérprete. A continuación listamos una breve descripción de lo que cada uno pretende probar.

Errores:

- test_error_break_fuera_iteracion.py
- test_error_continue_fuera_iteracion.py
- test_error_lexico_1.py
- test_error_lexico_2.py
- test_error_multiplicacion_string.py
- test_error_resta_string.py
- test_error_return_fuera_de_funcion.py
- test_error_sintactico_1.py
- test_error_sintactico_2.py
- test_error_suma_entero_string.py
- test_error_suma_float_string.py
- test_error_suma_long_string.py

Sentencias Control:

- test_while_sentence.py
- test_for_break_continue.py
- test_while_sentence.py

- test_if_sentence.py
- manual_test_if_else.py

Tipos Nativos:

- test_variables_sin_definir.py
- test_variables_y_tipos.py

Tipos Complejos:

- test_diccionario_clave_inexistente.py
- test_diccionarios.py
- test_tuplas.py
- test_listas.py
- test_comparaciones_diccionarios.py
- test_comparaciones_listas.py
- test_comparaciones_tuplas.py

Funciones:

- test_funciones_por_valor_y_referencia.py
- test_funciones_recursion_y_encadenadas.py
- manual_predefinidas.py

Operaciones:

- test_operaciones_bitewise.py
- test_operaciones_aritmeticas_numeros.py
- test_operaciones_strings.py
- test_operadores_binarios_test.py
- test_and_or.py

8.2. *Script para testear*

Como herramienta de apoyo, se implementó un mecanismo de scripts que ejecutan los casos de prueba, mostrando la salida de la ejecución y comparándola contra la salida que obtuvo python.

A modo de automatizar la generación de las salidas generadas por el compilador realizado en JAVA (que tendrán el sufijo _output_propio.txt) y por otro lado las salidas de Python (que tendrán el sufijo _output.txt), hemos creado un script llamado "generar_salidas_linux.sh" y "generar_salidas.bat". Este script utiliza el programa diff para comparar las ambas salidas. Es importante notar que aparecerán excepciones en la salida ya que hay test que deben tirar excepciones.

Ejemplo de ejecución (Windows):

C:\Compilador\tests\> generar_salidas.bat

Ejemplo de ejecución (Linux):

username@host:~/Compilador/tests$ ./generar_salidas_linux.sh

A su vez, existen también tests que requieren la interactividad con el usuario al ingresar datos a raíz de la utilización de la operación raw_input. Los nombres de estos últimos tests tienen el prefijo "manual_" ya que es necesaria la participación del usuario.

Ejemplo de ejecución:

- *C:\Compilador\tests\> java -jar ..\build\jar\Compilador.jar manual_test_if_else.py > salidas\manual_test_if_else.py_output_propio.txt*
  (e ingresar un valor del tipo entero)
- *C:\Compilador\tests\> python manual_test_if_else.py > salidas\manual_test_if_else.py_output.txt (e ingresar el mismo valor del tipo entero)*
- *C:\Compilador\tests\> diff.exe salidas\manual_test_if_else.py_output_propio.txt salidas\manual_test_if_else.py_output.txt*

9. *Errores conocidos*

A continuación, enumeramos los errores conocidos que tiene la solución implementada.

- A la hora de imprimir Diccionarios, al no tener orden el mismo, cuando se ejecuta “print d” siendo “d” un diccionario, puede imprimirse en otro orden al que python lo imprime.
- Dado “d1” y “d2” dos diccionarios, si ambos tienen la misma cantidad de elementos, no es posible comparar si uno es mayor/menor que otro.
- A la hora de redondear resultados de operaciones aritméticas como división, potencia, el redondeo queda distinto que en python.
- No se soporta que una función retorne dos elementos y sea asignados a variables distintas. Si podemos devolver tuplas.