Diseño de Compiladores

Implementacion de un compilador de Python

Cristiano Coelho - C.I.:4.298.259-3

Emiliano Conti – C.I.:5.033.483-5

Rafael Olivera – C.I.:5.044.399-9

# Introducción

El objetivo de este obligatorio será el diseño e implementación de un compilador del Lenguaje Python, utilizando las librerías JFlex y JCup para la implementación de las etapas de análisis y desarrollando un backend en Java que interprete el código intermedio generado.

En este informe se describirá el diseño elegido y los diferentes aspectos de la implementación de la solución desarrollada

# Análisis léxico

Para realizar el análisis léxico se utilizó la librería JFlex, la cuál recibe un archivo con la especificación del scanner, y genera un analizador lexicográfico, que se encargará de transformarel acrhivo de texto recibido en una secuencia de TOKENS.

El archivo implementado utiliza expresiones regulares para matchear los diferentes símbolos que puedan ocurrir en el archivo de entrada con TOKENS definidos que puedan ser facilmente reconocibles para realizar el análisis sintáctico.

En esta etapa es imperativo destacar diferentes manejos especiales que fueron tenidos en cuenta:

**Manejo de Strings:** Se definió un estado en el cuál se entra al reconocer una comilla, concatenando los símbolos que se van encontrando hasta encontrar la comilla de cierre

**Manejo de Strings concomillas triples:** En Python es posible definir Strings en varias líneas, utilizando comillas triples. Para este caso se procedió de la misma manera que con el manejo de Strings, pero ignorando los saltos de línea

**Manejo de Comentarios:** Se creó un estado de Comment, al reconocer el símbolo de Comentario, se ignora la entrada hasta reconocer un salto de línea

**Manejo de Indentación:** En Python, el anidamiento de bloques se realiza mediante indentación. Por tal motivo es necesario controlar la misma de una menra especial. Se utilizaron 2 símbolos INDENT y DEDENT que son insertados cuando aumenta o disminuye el nivel de indentación. Para implementar esto se utilizaron 2 variables globales que mantienen el nivel de indentación actual y el nivel de indentación de la línea que se está leyendo. Al reconocer un salto de línea, se entra a un estado en el cuál se utilizan los valores de estas variables parainsertar INDENTS Y DEDENTS cuando sea necesario. También al encontrar elfin del archivo (EOF) se realiza un control similar para imprimir los DEDENTs que sean necesarios.

**Preprocesamiento:** Considerando que un usuario puede insertar líneas vacías o solo con tabs o espacios, se implementó un preprocesador que elimina las líneas irrelevantes. Se tomó esta decisión para facilitar el trabajo tanto de Analizador léxico como del Sintáctico, debido a que la indentación juega un papel importante y líneas vacías o con tabs irregulares pueden acomplejizar innecesariamente el análisis

# Análisis Sintáctico y Semántico

Las siguientes etapas constan del análisis sintáctico y semántico. Para esto se utilizó la librería JCup, la cuál recibe un archivo con la especificación de la gramática y devuelve un parser escrito en Java.

La gramática comienza matcheando la secuencia de tokens como una lista de statements separados por saltos de línea. Luego, la misma se va bifurcando para matchear cada tipo de expresión permitida en Python. En particular, la primera decisión que se toma es si es una expresión simple(de una línea con una asignación, expresión, un print, etc.) o comupesta (if, while, for, definición de una función).

Cuando se matchea una producción de la gramática, se devuelve como resultado una instancia de la clase ParseResult, una clase que se definió para almacenar el número de línea y el valor obtenido de esa producción, así también como algunos argumentos opcionales. El valor que se retorna es una instancia de Bloque, una clase implementada que contiene las instrucciones generadas para esa producción(código intermedio)

Cada producción merece un trato especial a la hora de retornar su valor. Con este motivo se definió la clase RuleGenerator que contiene funciones específicas para diferentes producciones de la gramática. Estas funciones tienen comportamiento específico para cada tipo de expresión, se encargan de rellenar las tablas de símbolos, controlar errores semánticos(por ejemplo, variables no definidas) y generar el código intermedio. Finalmente, retorna un ParseResult con el número de línea y el código intermedio generado.

Se implementó además una clase Generador. Dicha clase contiene las tablas de símbolos, así también como operaciones para manejar las mismas, crear Bloques,etc.

En particular se utilizaron 2 tablas de símbolos: co\_names y co\_const, almacenando nombres y valores respectivamente.

# Generación de Código Intermedio

Como se mencionó en la etapa anterior, la clase RuleGenerator contiene las funciones que son llamadas desde la gramática para cada producción. Dichas funciones generan el código intermedio.

La representación elegida consta de una lista de instrucciones equivalentes al Python Bytecode. Se utilizó el comando ”dis.dis(código)” y documentación consultada por Internet para obtener todas las instrucciones requeridas y observar como maneja python estas instrucciones a bajo nivel

Dichas instrucciones serán interpretadas por el Backend, generando código java para cada una de ellas.

# Manejo de errores

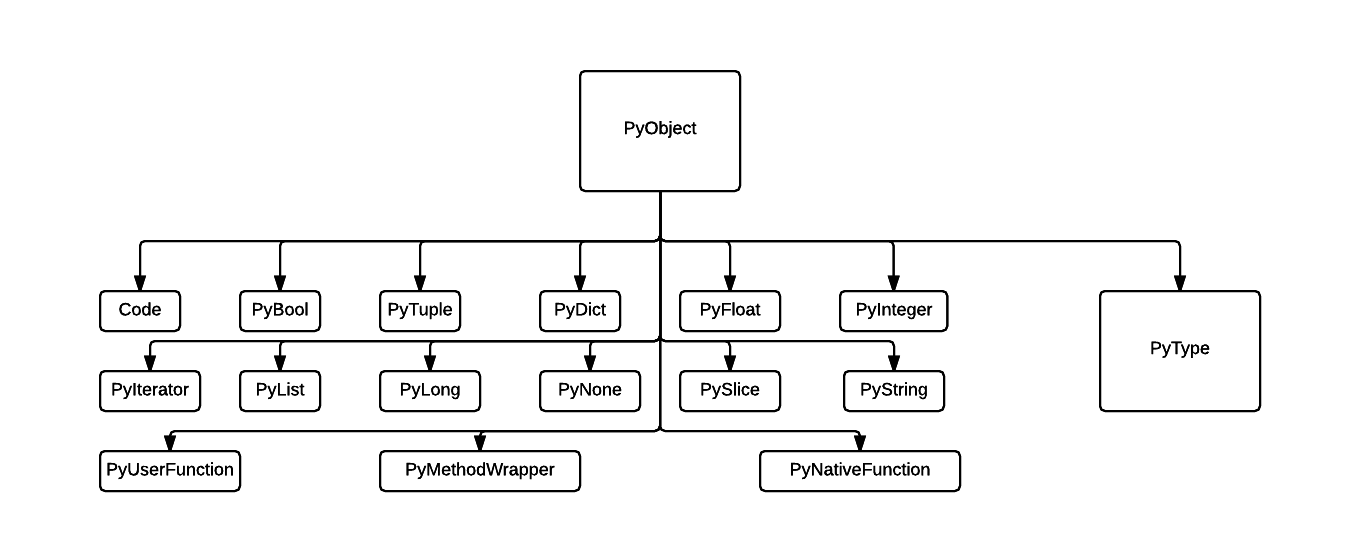
Para realizar el manejo de errores, se implementó una clase ParseStatus. Desde el main se llama al método ParseFile dentro de esta clase, y dentro se llama al Parser autogenerado que realizará la compilación. Esto permite mantener el estado del parsing y manejar correctamente los tipos y mensajes de error escupidos por el parser.

Por otro lado, tambien se manejan errores de ejecución o runtime, cada vez que se produce una operación ilegal, como invocar una función que no existe, o acceder a una variable que no fue definida, entre otros. Cuando un error de este tipo sucede, se imprime el stack trace con las lineas en donde se realizó cada llamada, y se termina la ejecución.

# Implementación de Backend

Para la implementación del BackEnd, se siguió una estructura similar a la de CPython, donde absolutamente todas las estructuras de datos son Objetos, incluyendo los tipos primitivos, funciones, y hasta clases.

Para ello, se puede observar el siguiente diagrama de clases.



En particular, todas las clases heredan de PyObject, la cual define todos los métodos que pueden llegar a existir para un objeto, como operaciones de suma, resta y multiplicación, instanciación, iteración, valores binarios (true/false), conversión a otros tipos (por ejemplo, de string a lista), obtener atributos, representación (print) y demás. Donde luego cada sub clase puede implementar o no dichas operaciones, lo cuál va a definir si una expresión puede ejecutarse, como por ejemplo, al hacer a + b, solo va a dar resultado si las clases de a y b implementan el operador +, donde en general los tipos numéricos van a implementarlo, como también string, y hasta listas, logrando así una gran abstracción de todas las funcionalidades, y logrando hacer el **chequeo de tipos** totalmente en runtime.

También, toda instancia de PyObject tiene un diccionario de atributos, en el cuál se almacenan otros PyObjects que pueden ser atributos o métodos del objeto.

Por otro lado, PyObject y todas las clases que heredan de esta, deben tener una referencia a su “clase”, la cuál será una instancia de PyType (que a su vez hereda de PyObject), por lo que todas las clases, son además instancias de otra clase, su “meta clase”. Es por esto, que la clase PyType es muy importante, ya que es la encargada de permitir definir todas las clases existentes en runtime (en nuestro caso, solo clases predefinidas ya que no permitimos definir clases), sus clases “bases”, como también manejar la instanciación de clases. Por ejemplo, al hacer int(“12”), en realidad, se está creando una intancia de int, a partir de una instancia de string, donde el hacer int(…) en realidad es llamar al constructor de int, y además int, es una instancia de PyType, la cual luego se encarga de instanciar el objeto.

Además, cada clase implementa sus propias funciones, las cuáles van a ser todas las posibles funciones predefinidas de esa clase, por ejemplo, dict.values().

Es importante destacar, que al acceder a un atributo o método de una instancia de un objeto, primero se busca su diccionario de atributos local, luego en el diccionario de atributos de su clase, luego de sus clases base, y finalmente se produce un error al no encontrarse.

Por otro lado, se tienen algunas clases particulares, como PyNativeFunction, PyMethodWrapper y PyUserFunction, las cuáles se encargan de almacenar código de funciones ejecutable, siendo esta última, la clase encargada de almacenar código definido por el usuario, y la primera, funciones predefinidas o nativas.

La clase PySingletons, es la encargada de unir todos los PyObject, PyType en una lógica que permite el correcto funcionamiento de todas estas clases y sus instancias.

Es así entonces, que todo el código ejecuta a través de la lógica implementada por PyObject y sus clases heredadas.

**Manejo de ambiente de ejecución:**

**En resumen:** Se utiliza memoria de tipo stack, con frames que almacenan datos del estado del a función ejecutando actualmente, un loop de interpretación e instrucciones de bajo nivel que manipulan el stack y objetos que representan todas las estructuras y funcionalidades del lenguaje.

La lógica responsable del manejo del ambiente de ejecución, se encuentra en el package com.pyjava.core.runtime.

En este package se define lo siguiente:

* **OpCode u operation codes:** Básicamente, un enumerado que define todas las posibles instrucciones que es capaz de interpretar el backend.
  + Las instrucciones estarán identificadas por un número.
* **Instruction:** Clase responsable de almacenar información de una única instrucción, en ella, se almacena la instrucción propiamente dicha, los argumentos (2 posibles argumentos, los cuáles inciden en como ejecuta la instrucción) y el número de línea al que corresponde la instrucción para poder realizar un reporte de errores detallado al producirse un error en runtime.
  + El parser entonces, entre otras tareas, genera instancias de esta clase para formar la lista de instrucciones o código intermedio a ejecutar.
* **Code:** Esta clase es la encargada de almacenar todo código ejecutable, definido por usuario, esto es, el código del módulo principal (archivo) como código de funciones definidas por usuario.
  + Esta clase entonces, almacena todos los nombres utilizados por el código (nombres de variables, funciones, atributos y demás), como también todas las constantes utilizadas (por ejemplo, al hacer a = 2, se almacena el nombre ‘a’, y un PyInteger con valor 2 como constante)
  + Almacena además una lista de instrucciones, nombre y argumentos de invocación en caso de ser código de una función.
  + Es el parser entonces, que se encarga de crear un objeto Code, con nombres, constantes, e instrucciones obtenidas luego de finalizar todo el proceso de parsing, el cuál es luego pasado al intérprete para que pueda ser ejecutado.
* **Frame:** Esta clase es la encargada de construir el stack de frames, para que sea posible la invocación de funciones, en particular, invocaciones recursivas.
  + En particular, cada frame tiene su propio stack interno, el cuál será utilizado por la función ejecutando actualmente.
  + Además, cada frame mantiene un diccionario de variables locales y globales, los cuáles mapean un nombre, a un valor (un PyObject), lo cual definirá el scope o alcance de la función ejecutando actualmente.
  + Por último, se maneja un contador de la instrucción actual, la cual se corresponde con la lista de instrucciones del código que está siendo ejecutado por este frame.
* **Estado:** Clase principal, responsable de mantener referencia a todos los tipos de datos que pueden ser utilizados, llamados ‘builtins’, en los cuales estarán tipos como int, long, string, y demás. Como también responsable de interpretar cada instrucción, mantener el frame que ejecuta actualmente, como también maniuplar el stack e implementar protocolos de invocación y retorno.
  + Es en esta clase, donde cada instrucción cobra significado y es interpretada, y según la instrucción, es el código que se ejcuta, como asignar variables, ejecutar funciones y demás.

**Proceso de interpretación:**

Con todo lo anterior definido, se resume el proceso de interpretación:

* Se crea una instancia de Estado, el cuál es inicializado con un Frame inicial, que contiene el Código principal obtenido por el proceso de parsing, se setea la instrucción actual en 0, y se inicia un buckle infinito, el cual obtiene la próxima instrucción a ejecutar, la interpreta, y repite el proceso, hasta llegar a una instrucción de fin de ejecución o un error de runtime.

Por otro lado, se resume el proceso de invocación y retorno:

* Al interpretar una instrucción de invocación o llamada de función, sucede lo siguiente:
  + Se leen los argumentos de la instrucción, y se sacan del stack todos los argumentos que debe recibir la función invocada.
  + Se lee del stack finalmente el objeto a invocar, ya que las funciones son instancias de PyObject
  + Luego, si la función invocada es una función nativa, en ese caso, se la invoca inmediatamente y se deja el resultado en el stack del frame actual, y aumenta en 1 la instrucción actual.
  + Por otro lado, si la función invocada es una función de usuario, se la invoca pasándole el estado actual, y esta realiza varios pasos:
    - Crea un nuevo frame, con dirección de retorno el frame actual, número de instrucción 0, y objeto de código el almacenado en el objeto de función.
    - Se validan argumentos, tanto posicionales como por nombre
    - Se agregan argumentos a las variables locales del frame actual
    - Se setean referencias a variables globales
    - Se pone en el estado el nuevo frame, y se retorna.
    - A partir de este punto, el loop de interpretación, obtiene la próxima instrucción ejecutar de un nuevo frame.
* Retorno:
  + Al interpretar una instrucción de retorno, se saca del stack del frame actual el valor del tope, se setea la referencia al frame actual en aquella almacenada como frame de retorno del frame actual. Se pone en el stack del nuevo frame el valor retornado, y se aumenta en 1 el número de instrucción actual.
  + Luego de este punto, se pasa a ejecutar instrucciones del nuevo frame.

Acceso y asignación de variables:

* Al acceder a una variable (mediante una instrucción particular, la cual tiene como argumento el índice en el diccionario de nombres del nombre a acceder), se busca primero en las variables locales del frame actual, luego en las variables globales, y finalmente en el diccionario de builtins (nombres globales del estado). En caso de no encontrarlo, se produce un error de ejecución indicando que la variable no existe.
* Al acceder a un atributo de un objeto (ya sea valor o función, son todos objetos), el proceso es similar, se busca el nombre que se quiere acceder en el diccionario de nombres, según el índice en el argumento de la instrucción, y se obtiene del stack el objeto del cual se quiere obtener su atributo, luego, se busca en ese objeto si existe el nombre, el cual busca primero en su diccionario de nombres, luego en el de su clase y clases base, y finalmente se produce un error si no se encuentra.
* La asignación de variables se hace de la misma forma, solo que siempre se guarda el valor en la tabla de variables locales del frame actual, por lo que no se pueden modificar variables globales, y el intentar hacerlo, simplemente ocultará del scope la variable global, ya que se siempre se encontrará primero la variable local.
* No se implementó la asignación de atributos a objetos ya que no existe manejo de clases.

# Errores conocidos

Luedo de realizar un exhaustivo testing al compilador, se encontraron diferentes errores menores comparando con la salida correspondiente:

* **Recursion infinita:** Controlada por el stack de java, no se hace control explícito.
* **Archivo vacio:** Ejecuta un rato y retorna error de sintaxis al final, no se controla del todo.
* **Print de listas dentro de listas:** Cuando una tupla, diccionario o lista contiene otra tupla, diccionario o lista como uno de sus elementos,elmismo se imprime simplificado, ya que si no pueden haber recursiones infinitas si un objeto tiene una referencia a si mismo, por lo que el print en este caso no funciona igual que el de python. Se puede arreglar implementando una especie de BFS o manteniendo variables globales para saber si un objeto ya fue visitado.
* **Longs infinitos:** Nuestra implementacion de python no soporta "longs" infinitos, como python, estan limitados al tamaño maximo de long de java.
* **Clausura de funciones:** Las funciones anidadas funcionan pero no se maneja de buena manera la clausura de las mismas. El único error que implica esto es que si dentro de una función se define una variable local, y luego se define una función dentro de la misma, el valor de la variable no es correctamente tomado por el scope de la segunda función. Esto ocurre solo con VARIABLES LOCALES, para variables definidas fuera de toda función funciona correctamente, se decidió no intentar resolver el error ya que es complejo y sería necesario hacer ciertas modificaciones a como se realiza la llamada de funciones y se setean las variables locales y globales del nuevo frame, y la implementación de definición de funciones anidadas no era un requerimiento.
* **Indentación con tabuladores:** Tal como dice la letra, la indentación se maneja solo con tabulación, a diferencia de Python en donde se soportan tabulaciones y espacios en blanco, como también múltiples tabuladores para una misma linea.
* **El manejo de slices:** Puede tener ciertas diferencias con los slices de python, en particular, la asignación de slices. En nuestro caso, no permitimos hacer asignación de slices con step negativo (ej: a[1:3:-1] = b) ya que se volvió algo complejo de implementar, debido a todos los índices negativos y a que se hizo lo posible para que los slices funcionen de la misma forma que en Python, considerando todos los casos bordes.