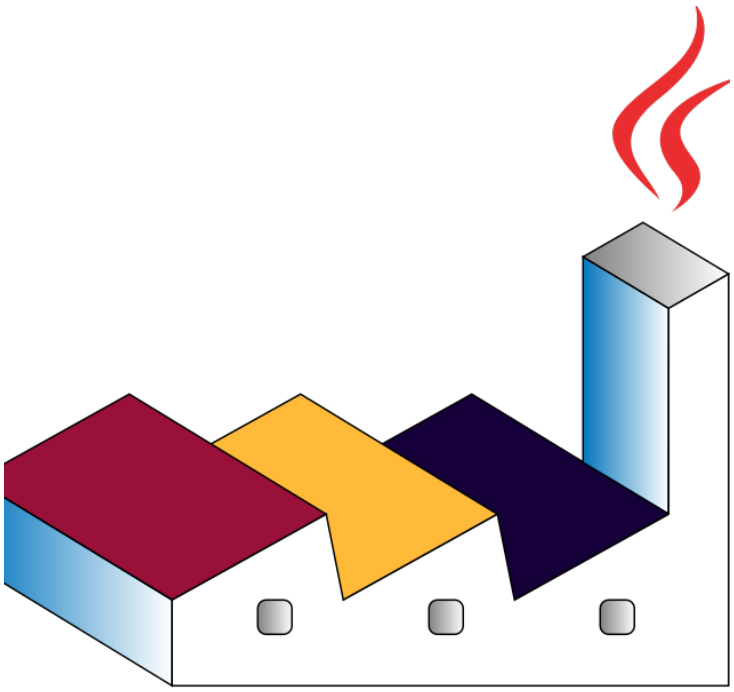
Autómatas, teoría de lenguajes y compiladores 72.39 - Curso 2022

Diseño e implementación de un lenguaje



Catino, Kevin Hiroshi (61643) - kcatino@itba.edu.ar

Chayer, Iván (61360) - ichayer@itba.edu.ar

Marengo, Tomas (61587) - tmarengo@itba.edu.ar

Mizrahi, Thomas (60154) - tmizrahi@itba.edu.ar

Fecha de entrega del informe: 23 de noviembre de 2022

Tabla de contenidos

Introducción

La idea del compilador provino de la familiaridad con el lenguaje Java y el extensivo uso que se le dió a los diagramas UML en materias como Programación Orientada a Objetos e Ingeniería de Software I. Si bien se utilizó PlantUML en numeradas ocasiones para diagramar un esquema de clases, y nos permitió conocer la gran utilidad de los llamados "Diagrams as code", siempre pensamos en la potencial utilidad que brindaría el poder traducir código escrito en UML a una serie de archivos con código del lenguaje de programación orientado a objetos correspondiente, con el objetivo de agilizar las etapas iniciales de meramente "traducir" el código de un lenguaje en otro.

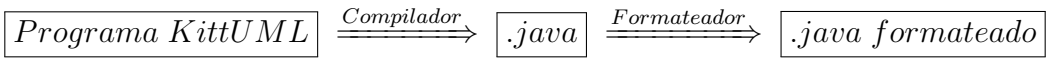
Así surgió la idea de KittUML, un compilador de KittUML (un lenguaje muy similar a PlantUML) a Java, que permite diagramar clases concretas, clases abstractas e interfaces con sus respectivos métodos y variables de clase/instancia. KittUML tiene la particularidad de que posee ciertas directivas (`comment`, `code` y `import`) que refieren al archivo de código final en Java, y entonces es posible generar archivos `.java` válidos para ser compilados. A partir de un único archivo de KittUML, es posible obtener una salida con múltiples archivos `.java`.

Consideraciones adicionales

Inicialmente, se debatió acerca de la posibilidad de ampliar el alcance de KittUML para poder incluir una serie de operaciones básicas como la suma, resta, concatenación de cadenas de caracteres, etc., de forma tal que el código introducido con la directiva `code` sea parseado en el proceso de compilación, y si bien no sería tan expresivo como cualquiera de los lenguajes de programación orientado a objetos, permitiría que un programa de KittUML sea completamente agnóstico al lenguaje de programación final y consecuentemente compilable a cualquiera de ellos.

Para poder llegar a un compilador durante la cursada, con la capacidad de generar archivos de Java con código medianamente complejo, se optó a que el código ingresado con la directiva `code` sea ignorado durante el proceso de compilación.

Por último, se decidió utilizar un formateador de código llamado "Google Java Format" que permite identar de forma correcta el código generado por el compilador. El usuario final no ve este paso intermedio, sino que directamente visualiza los compilados finales formateados al correr sl script.



Desarrollo del proyecto

Definición del lenguaje

Un programa de KittUML se delimita por las directivas `@startuml` y `@enduml`. Esto significa que cualquier texto por fuera de estos delimitadores se ignora durante la compilación.

Dentro del programa de KittUML, es posible definir clases (concretas o abstractas) e interfaces. Las clases pueden extender de otras clases e implementar interfaces, mientras que las interfaces pueden extender de otras interfaces:

```
@startuml

class PokemonClass {

}

class PokemonFireClass extends PokemonClass {

}

abstract class Pokemon<T> {

 }

interface Truchamon<T, V> extends Truchada<T> {

}

class Charizard extends Pokemon<PokemonFireClass> implements Iterable<
    Integer>, Truchamon<Faier, Integer>, String {

}

@enduml
```

Listing 1: Clases e interfaces

Decidimos permitir la implementación de interfaces y la extensión de clases que no están definidas dentro del esquema UML. Se valida que los nombres no colisionen (no pueden haber clases o interfaces con el mismo nombre, y las clases deben extender de otras clases o implementar interfaces, mientras que las interfaces sólo pueden extender de otras interfaces), pero no se verifica que las clases que se extienden o implementan se encuentran efectivamente dentro del programa. De esta forma permitimos casos de uso comunes como el hecho de que una clase implemente la interfaz `Comparable<T>`.

Una clase o interfaz puede opcionalmente incluir `imports` con la directiva correspondiente, pero estos deben situarse al inicio de la clase de manera obligatoria. Un `import` para ser válido no puede ser un nombre simple de caracteres del alfabeto, sino que debe ser al menos un grupo de dos "nombres"separado por puntos (ya que como mínimo debe haber un paquete dentro del cual se encuentra la clase a importar). Un detalle a resaltar es que el formateador de Java elimina las importaciones que no son utilizadas en el código. Si bien no es algo que planeamos desde el comienzo, no logramos encontrar la forma de remover esa funcionalidad en la herramienta. También es posible ingresar comentarios del UML (se ignoran para la generación del código Java) con el uso de la comilla simple (comentario inline) o con el uso de `\'` y `'\` (comentario multilínea).

```
@startuml

class Papafrita {
    'import: Import.buenas
    'import: java.util.ArrayList

    'comentario inline

    /'

    comentario multilinea

    '/
}

@enduml
```

Listing 2: Imports

Dentro de una clase podemos encontrar 3 tipos principales de elementos:

1. Comentarios de Java (con la directiva `comment`),
2. Variables,
3. Métodos.

Existen los modificadores de visibilidad privado (`-`), protegido (`#`) y público (`+`) y los modificadores de acceso `{static}` y `final`. También se puede denotar un método abstracto con `{abstract}`.

El compilador valida que las interfaces y clases abstractas no tengan métodos con modificadores `{static}` y `final` al mismo tiempo, y que, sólo las clases abstractas o interfaces puedan tener métodos abstractos (si un método es abstracto, entonces no puede tener

código asociado). En el caso de las interfaces, el compilador ingresa el modificador `default` si el método no es definido con `{abstract}` ni `{static}`. Los tipos de las variables pueden incluir genéricos que se anidan de forma recursiva.

```
@startuml
class Papafrita {
    'comment: Barcos en la marina
    # {static} final ArrayList constante /'code: 50; '/
    + {static} int metodo(A<A<B>> var) 'code: return -1;

  /'comment: comentario
                    2'/
}

@enduml
```

Listing 3: Métodos y variables

Como se puede observar en el ejemplo anterior, es posible introducir código a continuación de una método o variable (debe iniciar en la misma línea) para especificar código que aplica al elemento correspondiente. Las directivas `code` y `comment` pueden hacerse en forma inline o multilínea, pero `import` sólo se permite en la forma inline dado que una importación de una clase o paquete siempre ocupa una línea. Si un método es declarado no abstracto y no tiene un código declarado de su implementación, entonces el compilador lo autocompleta con un `return X;`, donde `X` corresponde a un valor predeterminado que es:

1. `0` para los valores numéricos,
2. `false` para `boolean`,
3. `null` para los objetos.

En el caso de que el método corresponda al constructor de la clase a la que pertenece, entonces el mismo se deja vacío con una llave de apertura y de cierre dado que un constructor en Java no retorna ningún valor.

Frontend

Para el frontend, debido a que aún no entendíamos completamente hasta qué punto era necesario definir la sintaxis del lenguaje (se nos dificultó la diferenciación entre la sintáctica y la semántica), creamos reglas de producción que ya validaban ciertas restricciones que podrían haberse realizado en la etapa de backend. Uno de los ejemplos que evidencia esto es en la definición de una clase, que podría haber sido una única producción, pero terminó

dividiéndose en 3 para validar que sólo una clase abstracta tenga métodos abstractos, que una interfaz no pueda usar `implements` o ser `abstract`, etc.

```
classDefinition: CLASS typeName[name] extends[ext] implements[imp]
    OPEN_BLOCK maybeEndlines inlineImportList[imports] classBody[body]
    CLOSE_BLOCK
                                    { $$ = ClassDefinitionGrammarAction(
    $name, $ext, $imp, $imports, $body); }
  ;

interfaceDefinition: INTERFACE typeName[name] extends[ext] OPEN_BLOCK
    maybeEndlines inlineImportList[imports] interfaceBody[body]
    CLOSE_BLOCK
                                    { $$ =
    InterfaceDefinitionGrammarAction($name, $ext, $imports, $body); }
  ;

abstractClassDefinition: ABSTRACT CLASS typeName[name] extends[ext]
    implements[imp] OPEN_BLOCK maybeEndlines inlineImportList[imports]
    abstractClassBody[body] CLOSE_BLOCK
                                    { $$ =
    AbstractClassDefinitionGrammarAction($name, $ext, $imp, $imports,
    $body); }
  ;
```

Listing 4: Producciones Bison de clases

Backend

Para el backend, implementamos primero la generación del árbol de sintaxis, seguido por la generación del string correspondiente a un código `.java`. Desarrollamos una librería llamada `buffer.c` que permite abstraernos de la administración de memoria necesaria para ir armando el string de los archivos finales. La interfaz de la misma se muestra a continuación:

```
bufferADT init_buffer(char * file_name);
void destroy_buffer(bufferADT buffer);
void generate_file(bufferADT buffer);
void write_buffer(bufferADT buffer, char * s);
char * get_current_string_buffer(bufferADT buffer);
```

Listing 5: Interfaz buffer.h

Si bien inicialmente necesitábamos únicamente la función `generate_file()` para poder generar el archivo `.java` correspondiente, terminamos creando una segunda función llamada

`get_current_string_buffer()` que nos permitió utilizar el tipo abstracto de datos en otros contextos donde se requería ir construyendo una cadena de caracteres y aprovechar las funciones ya desarrolladas para interpretar nodos del árbol de sintaxis y generar un string en base a ello.

Como etapa final, agregamos una tabla de símbolos con entradas como las siguientes:

class_id	name	type
1	Empanada	class
1	Empanada()	class_elem
1	Empanada(int, String)	class_elem
1	metodo(int)	class_elem
2	Iterable	class & interface
2	variable_1	class_elem

Sabíamos que debía haber una diferenciación entre las clases y las interfaces, pero al mismo tiempo una de las restricciones que debíamos imponer es que una clase no puede tener el mismo nombre que otra clase, pero tampoco el de otra interfaz, y viceversa. Por eso decidimos darle un flag de tipo `class` a todas las clases e interfaces, y agregar un flag de `interface` únicamente a las interfaces. De esta forma con una máscara era fácil definir si una entrada de la tabla colisionaba con otra o no.

Para el caso de los métodos, decidimos guardar la firma completa del mismo, dado que en Java (y en muchos otros lenguajes orientados a objetos) es posible la sobrecarga de un mismo nombre, permitiendo su repetición siempre y cuando la firma entre ellos difiera (la firma se compone por los tipos de parámetros de los métodos en el orden en el que se definen).

Dificultades encontradas

A continuación listamos algunas de las dificultades encontradas durante el desarrollo del proyecto:

- No comprendíamos por qué los imports no se visualizaban en el archivo final, hasta que descubrimos que en el archivo "crudo"previo al formateo aparecían, y era Google Java Format quien borraba los imports sin utilizar dentro del código.
- No comprendíamos por qué obteníamos ciertos valores en los enums que parecían imposibles, hasta que encontramos que erróneamente hacíamos la suma de dos flags, en lugar de una operación OR bit a bit.
- Hubo que investigar las restricciones que pone Java sobre las declaraciones de méto-

dos y variables. Por ejemplo, un método en una interfaz puede no tener implementación (lo más común), o ser declarado como "defaultz traer una implementación por defecto. Por el otro lado, un método estático debe sí o sí tener código, esté en una clase o en una interfaz.

Futuras extensiones y/o modificaciones

Si bien KittUML actualmente funciona como compilador para archivos de Java, teniendo en cuenta que los diagramas de clase son agnósticos al lenguaje de programación, nos gustaría desarollar configuraciones del compilador para diferentes lenguajes de destino como C#, Ruby, Python, entre otros.

Uno de los problemas que encontraríamos en caso de querer ir por este camino es que la directiva de `code` actualmente se procesa como un único token en forma de string, y en ningún momento se analiza su sintaxis o semántica dentro del proceso de compilación. Es por eso que en la actualidad, se confía en que el usuario introducirá código compatible del lenguaje objetivo, con la ventaja de que la complejidad del fragmento de código es ilimitada, pero con la desventaja de que su contenido podría poseer errores que no se capturan durante la compilación.

Si quisiéramos hacer que un único código de KittUML sea compilable a varios lenguajes de programación, tendríamos que desarrollar un lenguaje de programación propietario de KittUML que permita realizar un conjunto de operaciones básicas propias de un lenguaje de programación orientado a objetos, y que pueda ser parseado y consecuentemente traducido al lenguaje de programación que se desee. Sería un desafío poder plantear un lenguaje lo suficientemente expresivo como para ser útil para un usuario, ya que quizás con los tipos de datos String, Int y Double no sea suficiente. Creemos que la posibilidad de referenciar a la entidad (el uso de `this`), al igual que la posibilidad de poder utilizar clases definidas dentro del UML como tipos de datos dentro del código serían cualidades deseadas.

Por último, como Pedro McPedro fue una parte divertida de las correcciones, decidimos conservar su licencia para el recuerdo.

Bibliografía

No se utilizaron recursos por fuera de lo provisto por la cátedra.