Trabajo práctico especial
Diseño e implementación de un lenguaje
2022

Autómatas, teoría de lenguaje y compiladores

Integrantes:

- Budó Berra, Gaspar Andrés
- Hadad, Santiago
- Squillari, Bruno Omar
- Zimbimbakis, Facundo

Contenido

Introducción con idea subyacente y objetivos del lenguaje 2

Descripción de la gramática/sintaxis del lenguaje diseñado 3

Descripción del desarrollo del proyecto y de las fases del compilador 7

- Frontend 7
- Backend 7

Dificultades a la hora de desarrollar el proyecto o alguna de sus partes (incluso problemas al concebir la idea original) 8

Futuras extensiones y/o modificaciones, indicando brevemente la complejidad de cada una 8

Referencias 9

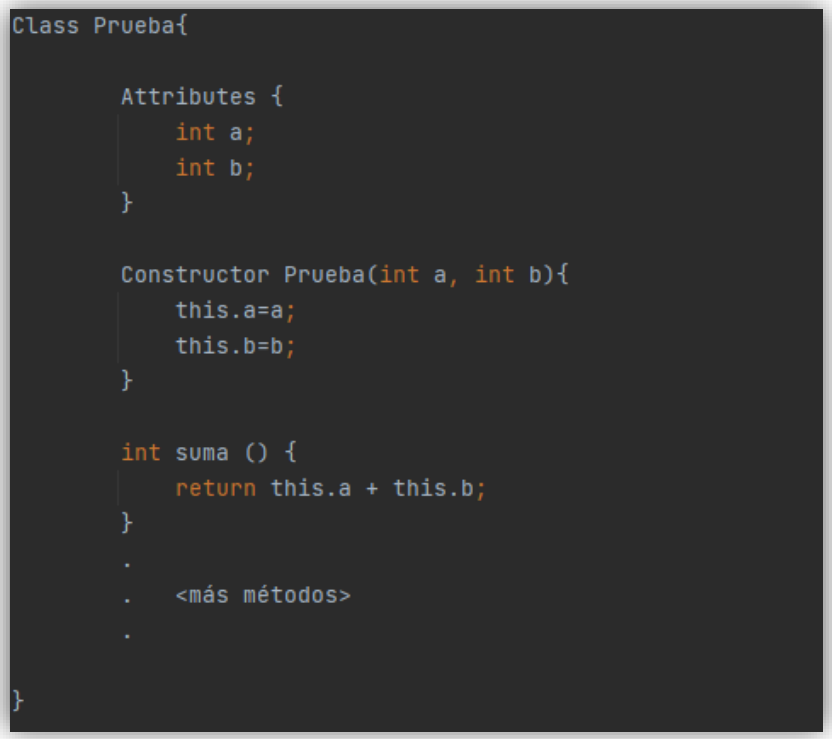
Introducción con idea subyacente y objetivos del lenguaje

La idea era hacer un lenguaje de programación similar a C pero utilizando el paradigma de programación orientado a objetos. Si bien sabíamos de la existencia de C++, nuestra idea era tratar de solucionar el problema de este mismo: es muy complejo porque permite utilizar las operaciones más avanzadas del lenguaje C, como son los punteros y manejo de memoria.

Esto decidimos llevarlo a cabo mapeando las clases y sus atributos con una struct que los represente. A partir de esto, se pensó la forma de implementar el constructor. El mismo debía retornar un puntero a una struct declarada anteriormente con los atributos que se hayan pasado por parámetro, ya definidos. Luego, cada método de una determinada clase tiene al menos un parámetro, el puntero a la estructura previamente creada por el constructor, el cual se agrega a la hora de la traducción.

Descripción de la gramática/sintaxis del lenguaje diseñado

Primero que nada, al ser un lenguaje pseudo-orientado a objetos debimos incluir la posibilidad de crear clases. Cada una de ellas puede estar conformada por sus atributos, su constructor (indispensable) y sus métodos. Como por ejemplo:



```
Class Prueba{

        Attributes {
            int a;
            int b;
        }

        Constructor Prueba(int a, int b){
            this.a=a;
            this.b=b;
        }

        int suma () {
            return this.a + this.b;
        }
        .
        .   <más métodos>
        .

}
```

En el ejemplo se pueden detectar las palabras reservadas para llevar a cabo nuestro lenguaje:

➔ **Class**: Palabra reservada para crear una clase.

- **Attributes**: Palabra reservada para incluir los atributos de una respectiva clase. Esta palabra solo puede utilizarse dentro de una clase y debe ser el primer elemento de la misma. No es obligatorio en la definición de una clase.

- **Constructor**: Palabra reservada para el constructor de una clase. El constructor es la función que será utilizada para instanciar cualquier clase. Es obligatorio, solo puede utilizarse dentro de una clase y se encuentra a continuación de los atributos.

- **this**: Palabra reservada para hacer referencia a un atributo

dentro de la clase. Solo puede utilizarse dentro de una clase.

Como podemos ver en el ejemplo, cada atributo de una clase que se quiera utilizar debe contener "this." como prefijo. Tomamos la decisión de implementarlo de esa forma para evitar confusiones al momento en que un parámetro tenga el mismo nombre que un atributo. Podemos apreciar como evitamos esta confusión de manera clara en el constructor del ejemplo anterior.

Luego, cada programa debe contener un main al igual que en C. La función main será la función que primero se ejecute. Un ejemplo podría ser:

```
int main(){
    Prueba r= new Prueba(3,2);
    r.suma();
    int integer = r.a;
}
```

En el ejemplo podemos notar las palabras reservadas:

- **main**: Palabra reservada para la función principal del programa. En este caso se mantuvo la idea del lenguaje C.
- **new**: Palabra reservada para instanciar una clase creada anteriormente. Por debajo, se llama al Constructor dentro de esa clase al igual que Java.

En la función main podemos instanciar las distintas clases con el find e acceder a sus atributos o utilizar sus métodos. La sintaxis se mantiene idéntica a la de Java al igual que en el caso en que se quiera desreferenciar un atributo.

A continuación, se implementó la posibilidad de heredar a partir de una clase (herencia) o también, que una clase sea parte de los atributos de otra (composición). Sintácticamente, ambos se pueden diferenciar del otro, pero por detrás se manejan de forma similar.

Un ejemplo de herencia podría ser:

```
Class Cuadrado{
        Attributes {
            int lado;
        }

        Constructor Cuadrado(int lado){
            this.lado = lado;
        }

        int area () {
            return this.lado * 2;
        }
        int perimetro(){
            return this.lado * 4;
        }
}


Class Cubo extends Cuadrado{
        Attributes {
        }

        Constructor Cubo(int lado){
            this.lado=lado;
        }

        int area() {
            return this.lado * 2 * 6;
        }

        int sumaAristas(){
            return this.perimetro() * 3;
        }
}
```

En este ejemplo podemos distinguir una nueva palabra reservada utilizada:

- ➔ extends: Palabra reservada para heredar atributos y métodos de una clase ya definida anteriormente.

Por el otro lado, un ejemplo con composición de clases podría ser:

```
Class Fibonacci{

    Attributes {
            int a;
            int b;
        }

        Constructor Fibonacci(){
          this.a=1;
          this.b=0;
        }


        int next () {
         int aux = this.a;
         this.b = this.a;
         this.a = this.a + this.b;
         return aux;
        }


}

Class Prueba{

    Attributes {
        Fibonacci f;
    }

    Constructor Prueba(Fibonacci f){
      this.f=f;
    }
}
```

En este ejemplo no tenemos nuevas palabras reservadas.

Por último, los tipos de datos primitivos se seleccionaron los principales utilizados en C: int y char con sus respectivos arrays.

Descripción del desarrollo del proyecto y de las fases del compilador

Frontend

En esta etapa, se tokeniza el programa de entrada mediante el uso de flex. Una vez segmentada la entrada, es más fácil analizar si la misma corresponde a un programa sintácticamente correcto en nuestra gramática. La misma, se encuentra definida en bison, indicando las producciones que son tomadas como válidas. En esta etapa únicamente se rechazan los programas que no pertenecen al lenguaje generado por la gramática mencionada.

Backend

Una vez aceptada una cadena, se comienza a procesar la misma. Para esto se genera el árbol de derivación correspondiente a dicha entrada. El mismo es generado de manera ascendente por medio de las acciones de bison. Este árbol sirve para un posterior análisis semántico que se efectúa. A su vez, el árbol es utilizado para realizar la traducción de la entrada al lenguaje C.
El análisis semántico se realiza mediante una tabla de símbolos creada a partir del nodo raíz del árbol. Dicha tabla almacena las entidades del programa y su tipo. Las entidades consideradas son las clases, los atributos de cada clase, los métodos de cada clase, y , por último, las variables de dichos métodos y del main. Este analizador chequea que todas las operaciones sean válidas, es decir, que los tipos de las asignaciones sean los correctos, que las variables que se usan estén declaradas, que las clases estén declaradas. El analizador se crea de manera recursiva recorriendo el árbol de vuelta, y recién cuando el programa es aceptado por el mismo, se genera la correspondiente traducción.
Funcionalidades del análisis semántico:

- Uso de variables previamente declaradas.
- Asignaciones con tipos de datos correspondientes.
- Llamados a métodos de una clase y sus respectivos parámetros.
- Parámetros de funciones de C. (No se chequea el tipo de retorno de estas por no estar declaradas en nuestro programa de entrada).
- Extensiones a clases existentes.
- Uso de atributos correctos.
- Coincidencia entre el return y el tipo declarado en el método.

Dificultades a la hora de desarrollar el proyecto o alguna de sus partes (incluso problemas al concebir la idea original)

Uno de los principales problemas encontrado fue la determinación de todas las posibles maneras que se puede acceder a otra variable. Por ejemplo, atributo de objeto, desreferenciación de atributo de objeto, atributo de la desreferenciación de atributo de objeto entre otros. Se intentó abarcar la mayor cantidad de opciones mediante la recursividad, aunque hay algunos accesos complejos que no se tuvieron en cuenta.
Otro problema que encontramos fue que definimos las reglas del bison de manera extensa por lo que luego generaba conflictos de shift-reduce. Para solucionarlo, tuvimos que acudir a numerosas eliminaciones y reutilizaciones de las reglas.
La herencia en un principio no resultaba fácil entender por dónde encarar este objetivo. Pero luego mediante el uso de la lista de símbolos y referencias cruzadas entre las distintas clases se logró poder validar este concepto e implementarlo.

Futuras extensiones y/o modificaciones, indicando brevemente la complejidad de cada una

En el futuro teníamos pensado extender las posibilidades de nuestro lenguaje.
Una de las extensiones que deberíamos implementar es un garbage collector básico, el cual se encarga de liberar toda la memoria utilizada en el programa traducido. Esto se puede llevar a cabo mediante un llamado a la función free() al final del main(). Simplemente se debe hacer una llamada a esta función por cada nueva instancia de una clase que se haya encontrado. La complejidad de esta extensión es justamente, la particularidad de guardar todas las instancias de las clases creadas, para luego hacer la liberación de memoria correctamente.

Otra idea que nos gustaría incorporar al lenguaje es el control de visibilidad al igual que Java mediante las palabras reservadas private y public. Esto podría hacerse posible gracias al uso de una tabla de símbolos para detectar si es adecuado el sector en el que se está llamando a un determinado método/atributo.

Referencias

Flex-Bison example by agolmar

ANSI C Yacc grammar by jutta@pobox.com

(2009) Levine - flex & bison

(2009) Segal - Writing Your Own Toy Compiler Using Flex, Bison and LLVM

(2021) GNU Bison - Defining Language Semantics