Informe del Lenguaje de Programación ~~RWLZ~~ c-minor

1. Introducción

RWLZ inició como un lenguaje inspirado en la sintaxis de C#, con la intención de ser compatible con el ecosistema .NET. Sin embargo, debido a problemas técnicos, cambios en el código a referenciar durante el semestre y limitaciones de tiempo, el proyecto tuvo que cambiar sus objetivos y terminó convirtiéndose en **c-minor**, que es la versión entregada como proyecto final.

2. Motivación y Objetivos

La idea original de RWLZ era complementar la librería *Fisobs [https://github.com/Dual-Iron/fisobs]*, permitiendo crear criaturas personalizadas a través de un lenguaje propio. El enfoque era usar una sintaxis similar a C#, pero con paradigma imperativo para acelerar la creación de la lógica esencial. La meta era generar código C# automáticamente o directamente un DLL insertable en el videojuego “Rain World”.

El objetivo final de este proyecto consistía en permitir crear una criatura básica heredada de clases ya existentes del juego. Sin embargo, debido a incovenientes como cambios en el código base del juego, incompatibilidades y la complejidad del proyecto en el tiempo disponible, esta versión tuvo que descartarse. El enfoque se redirigió entonces hacia **c-minor**, un lenguaje más pequeño pero funcional, diseñado específicamente para cumplir con los requerimientos del proyecto final.

Actualmente tiene la capacidad de utilizar cualquier código con la sintaxis básica de C.

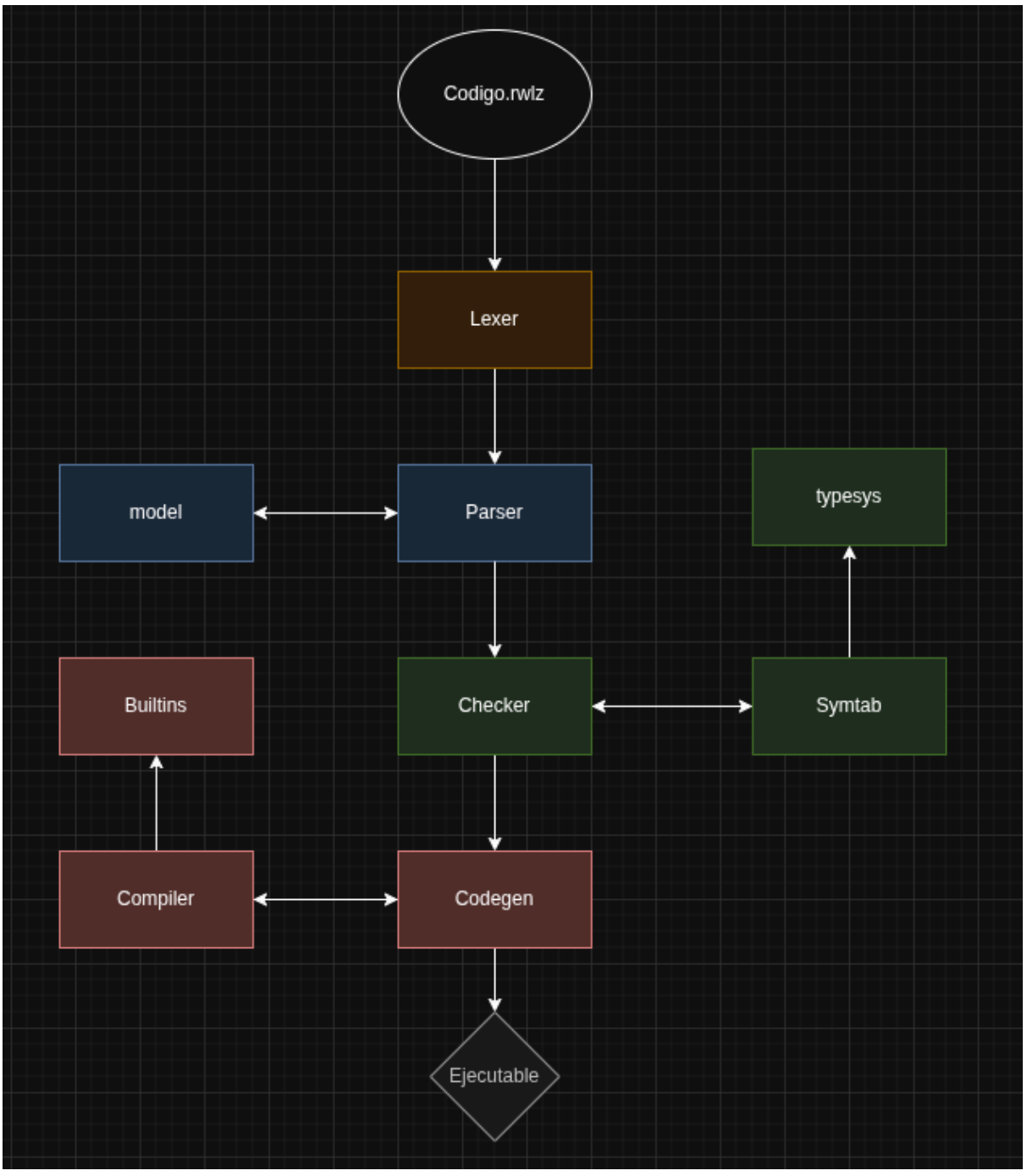
3. Diseño del Lenguaje

c-minor conserva algunas ideas de RWLZ, pero con un alcance más reducido.

Características heredadas:

- Programación imperactiva
- Enfoque centrado en funciones
- Soporte para funciones predefinidas
- Sintaxis inspirada en C# y C, con algunas facilidades estilo C (por ejemplo, `print`)

Esta es la estructura del compilador



4. Lexer

El desarrollo del lexer fue relativamente sencillo, basado en ejemplos previos. Aunque algunos tokens quedaron sin uso debido al cambio en el diseño, el lexer final se ajusta bien a las necesidades de b-minor. Fue necesario añadir operadores binarios y el operador módulo (%), que inicialmente no estaban contemplados.

Orden de reconocimiento:

1. Operadores compuestos (`+=`, `-=`, `*=`, `/=`, `++`, `--`)
2. Operadores de comparación (`==`, `!=`, `<=`, `>=`)
3. Operadores lógicos (`&&`, `||`)
4. Operadores simples (`+`, `-`, `*`, `/`, `%`, `=`, `<`, `>`, `!`)
5. Palabras clave (`int`, `float`, `if`, `while`, etc.)
6. Literales
7. Variables, identificadores

`Ejemplo:`

```
None
[BepInPlugin("RainLizard", "Lizard Mod", "1.0")]
/* Archivo de prueba */

bool Move(int speed, float power) {
    int steps = 10;
    print("moviendo al jugador");
    if (steps > 5) {
        return false;
    } else {
        return true;
    }
}
```

5. Gramática

La gramática fue más compleja que el lexer. Aunque el ejemplo base ayudó bastante, fue necesario expandirlo para incorporar nuevas funciones y manejar correctamente expresiones, operadores y estructuras.

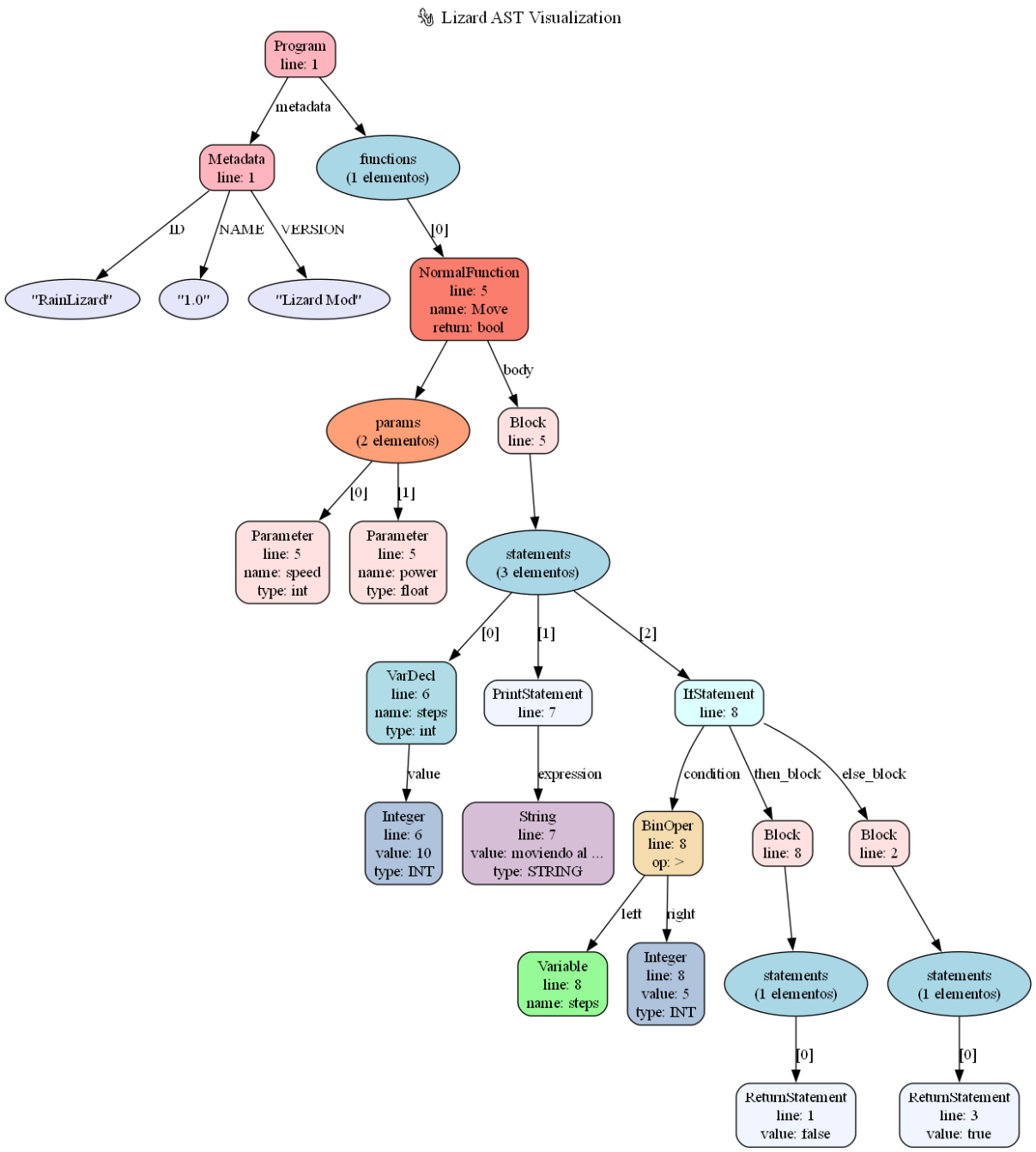
Un ejemplo de una parte que quedó obsoleta es la declaración de metadatos:

```python
@_('metadata_decl decl_list')
def program(self, p):
    # Metadatos opcionales (dependían de RWLZ)
    return _L(Program(metadata=p.metadata_decl,
functions=p.decl_list), p.lineno)
```

Finalmente, estos metadatos quedaron como opcionales, y en c-minor ya no se utilizan.

Para construir el parser se implementó un sistema de nodos que facilita la creación del AST. El nodo principal (`Program`) agrupa declaraciones, funciones, expresiones y sentencias.

Además, se integró **graphviz** para visualizar el árbol. Usando `--png` se puede generar la imagen del árbol, y con `--dot` ver la salida cruda del parser.



6. Analizador semántico

El analizador semántico es una de las partes centrales del compilador de **c-minor**, pues se encarga de comprobar que el programa tenga sentido más allá de la sintaxis. Mientras que la gramática asegura que el código *esté bien formado*, el análisis semántico verifica que *sea correcto* según las reglas del lenguaje.

Para lograrlo, se utilizan dos módulos fundamentales:

- **La tabla de símbolos (`symtab.py`)**
- **El sistema de tipos (`typesys.py`)**
- **El comparador de tipos (checker.py)**

Estos trabajan en conjunto durante el recorrido del AST.
La tabla de símbolos se puede visualizar usando `--sym`, mostrando cada símbolo y su tipo.

7. LLVM

Este se encarga de obtener el árbol hasta verificado, y empieza a generar el lenguaje intermedio y luego compilarlo dependiendo del sistema operativo. En esta parte uya estaba decidido generar el código en LLVM y no para la máquina virtual .net (aunque ya tengo una idea general de cómo funciona el proceso, no me queria arriesgar a otra incompatibilidad).

Se creó el siguiente mapa para el almacenamiento de los tipos de datos.

```
self.type_map = {
            BaseType.INT: ir.IntType(32),
            BaseType.FLOAT: ir.DoubleType(),
            BaseType.BOOL: ir.IntType(1),
            BaseType.CHAR: ir.IntType(8),
            BaseType.VOID: ir.VoidType(),
            BaseType.STRING: ir.IntType(8).as_pointer()
        }
```

y se detecta para compilar a la plataforma esperada

El código recorre el árbol ast para traducirlo a lenguaje intermedio y luego compilarlo, ejemplo del lenguaje intermedio se puede hacer usando el argumento “--compile”:

8. Conclusiones

Esto ha sido una buena experiencia, a pesar de que me tocó hacer el proyecto en solitario, y el agotamiento del tiempo, logre resolver y entender mejor cómo funcionan los lenguajes por dentro, y me ayudara en mis proyectos personales al entender mejor el comportamiento interno de lenguajes como c#, para poder aprovechar al máximo sus capacidades y poder hacer técnicas más avanzadas de ingeniería inversa en un futuro, ahora que tengo los fundamentos de cómo funcionan los lenguajes de programación. Gracias por este reto, profesor :D