Proyecto Odin.

John Mauris López Ramos.

MatCom. C-121.

Introducción.

Este proyecto se enfoca en el desarrollo de un compilador para un Lenguaje de Dominio Específico (DSL) diseñado para la creación de cartas de juego. El objetivo es optimizar la definición de las propiedades y habilidades de las cartas de manera clara y accesible para los desarrolladores de juegos.

Para usar el compilador debes importar el archivo Odin.dll a tu proyecto y en el código en C# agregar '*using Odin;'.* Para crear la carta debes emplear *Run.RunCode("Example Text…")* y pasarle el código en DSL como una cadena de caracteres y mediante *Run.CardsCreated* recibes una lista de las cartas creadas, luego para poder correr el efecto usa *Run.RunEffect(cardName, gameState)* que recibe como parámetros el nombre de la carta jugada y el estado del juego actual como una instancia de la clase *GameState*, puedes acceder a los errores que ocurran durante las diferentes etapas de la compilación o corrida a través de *Run.Errors*.

Características Principales del DSL.

La sintaxis del DSL está diseñada para ser clara y minimalista.

El DSL utiliza un sistema de tipado implícito, donde no es necesario declarar el tipo de datos de cada atributo, ya que el compilador lo infiere automáticamente. Esto hace que la escritura de cartas sea rápida y sin complicaciones.

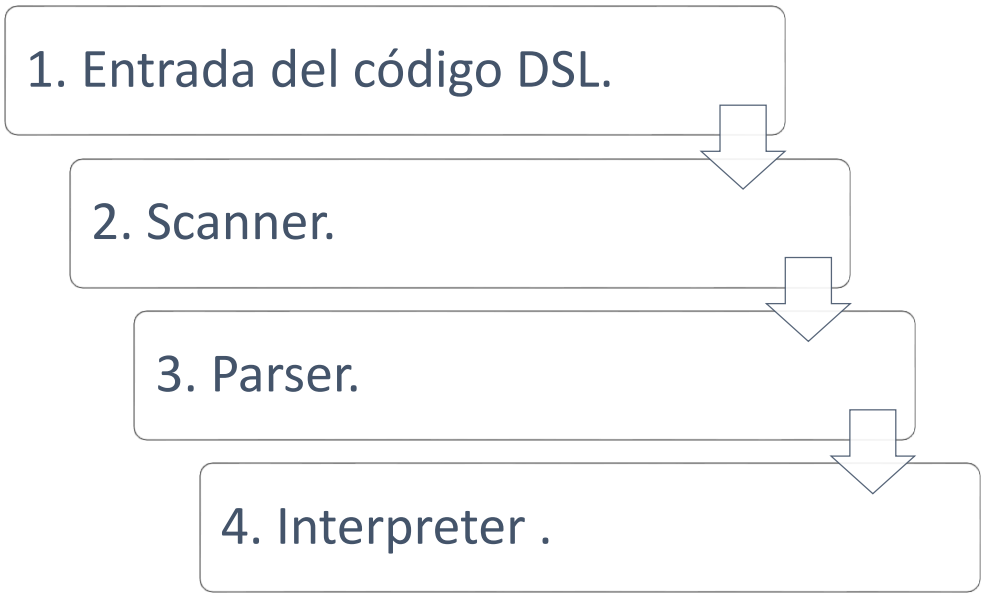
El compilador realiza validaciones tanto sintácticas como semánticas durante la fase de análisis. Se asegura de que las cartas definidas cumplan con las reglas establecidas, evitando la creación de cartas con valores inválidos o malformaciones en la sintaxis.

Ejemplo:

```
effect {
    Name: "Senuelo",
    Action: (targets, context) => {
        maxPower = -1;
        for target in targets{
            maxPower=max(maxPower, target.Power);
        };
        CARD = targets.Find((unit) => unit.Power == maxPower)[0];
        context.Field.Remove(CARD);
        context.Hand.Push(CARD);
    }
}

card {
    Type: "Oro",
    Name: "Odin",
    Faction: "Rick" @@ "Sanchez",
    Power: (4 + 2 * 5 / 3) ^ 1,
    Range: ["Melee", "Ranged", "Siege"],
    OnActivation: [
        {
            Effect: "Senuelo",
            Selector: {
                Source: "field",
                Single: false,
                Predicate: (unit) => unit.Type == "Oro"
            }
        }
    ]
}
```

Diagrama de bloques del compilador.



1. El compilador comienza con el código fuente escrito en DSL, que en este caso es el formato en el que los usuarios definen las cartas del juego.
2. En esta etapa, el código fuente del DSL se descompone en sus componentes más básicos llamados tokens. Cada token representa una unidad mínima del lenguaje, como palabras clave (card), nombres de variables (cartas, var), valores (5, true, ”Oro”), y símbolos ({, }).
3. El parser toma la lista de tokens generada por el scanner y construye un árbol de sintaxis abstracta. Este árbol refleja la estructura jerárquica del código DSL, asegurando que siga las reglas gramaticales definidas para el DSL.
4. Después de que el parser genera el AST, el intérprete toma este árbol y lo evalúa directamente. Interpreta el significado del código y realiza las acciones definidas. Simula la creación de cartas para el juego y ejecuta los efectos.

Principales desafíos.

- Errores de sintaxis y manejo de excepciones:
  Se implementó un sistema de manejo de errores durante todas las etapas que no solo identificaba el error, sino que también proporcionaba mensajes detallados sobre la naturaleza del problema.
- Coherencia semántica:
  Se integró un análisis semántico que validaba tipos y valores, por ejemplo, verificando que *Power* y *Name* fueran enteros o strings y que los tipos correspondieran a los tipos definidos en el juego.
- Operadores aritméticos:
  Se implementó el manejo de operaciones aritméticas de tipo suma(+), resta(-), multiplicación(*), división(/), exponenciación(~)[calculada en complejidad logarítmica], resto(%), and(&), or(|), xor(^), concatenación(@), concatenación con espacio intermedio(@@) y bit shiffting(<< y >>).
- Operadores de incremento o decremento:
  Se implementaron los operadores (++) y (--) por delante y detrás de statements, ejemplo: ++var o var--.
- Asignación y comparación:
  Se implementó la asignación acompañada de operadores aritméticos o sola (=, +=, -=, *=, /=, %=, &=, |=, ^=, @=, @@=, <<=, >>=).
- Funciones primitivas:
  Se implementaron funciones primitivas como log(a, b), rand(a, b), max(a, b) y min(a, b).
- Patrón Visitor:
  Se implementó el patrón de diseño Visitor, donde cada nodo del árbol de sintaxis (por ejemplo, *card*, *Type* o *Name*) tiene un método accept(visitor) que acepta un visitante (visitor). El visitante luego aplica la operación necesaria en el nodo. El patrón Visitor facilita la adición de nuevas fases en el compilador sin alterar la estructura original del árbol, haciendo el código más extensible y fácil de mantener.

Estructura del código fuente del compilador.

- NormalizeInput.cs:
  Elimina de la entrada caracteres innecesarios como '\r'.
- Token.cs:
  Define la estructura básica de un token.
- TokenType.cs:
  Contiene todas las palabras claves y símbolos que pueden ser usados en el DSL como *enum*.
- TokenDictionary.cs:
  Guarda en diccionarios las palabras clave asociadas a los *enum*.
- ErrorReporter.cs:
  Maneja los errores y los devuelve de una manera comprensible para el usuario **sin** el uso de *ThrowException()* de C#.
- ErrorPhrases.cs:
  Contiene frases que pueden ser devueltas junto a los errores.
- Scanner.cs:
  Implementa el *Scanner* explicado.
- AST:
  Contiene los nodos del AST.
  - Class:
    - Class.cs:
      Define la clase abstracta *Class*.
    - CardClass.cs:
      Hereda de *Class* y define la estructura de una carta necesaria para la interpretación.
    - EffectClass.cs:
      Hereda de *Class* y define la estructura de un efecto necesaria para la interpretación.
  - Expr:
    - Expr.cs:
      Define la clase abstracta *Expr*.
    - Binary.cs:
      Define la estructura de las operaciones binarias.
    - Call.cs:
      Define la estructura de las llamadas a funciones.
    - Find.cs:
      Define el método *Find()* en *Lists*.
    - Get.cs:
      Define la obteción de valores en propiedades.
    - Grouping.cs:
      Define la agrupacion de expresiones entre paréntesis.

        - Index.cs:
          Define la forma de indexar en *Lists*.
        - Literal.cs:
          Define el almacenamiento de literales de tipo *Number*, *String* y *Bool*.
        - PostOper.cs:
          Define un nodo para los operadores aritméticos de incremento y decremento situados posteriores a una variable o propiedad.
        - PreOper.cs:
          Define un nodo para los operadores aritméticos de incremento y decremento situados antes de una variable o propiedad.
        - Set.cs:
          Define la modificaion de valores en propiedades.
        - Unary.cs:
          Define las operaciones unarias ! y -.
        - Variable.cs:
          Define las variables.
    - Method:
        - Method.cs:
          Define la clase abstracta *Method*.
        - Action.cs:
          Define la estrucutra del *Action* de *effect*.
        - Effect.cs:
          Define la estructura de *Effect* en *card*.
        - OnActBody.cs:
          Define la estructura compuesta por *Effect*, *Selector* y *Predicate*.
        - OnActivation.cs:
          Define la estructura de *OnActivation*, es padre de *OnActBody*.
        - Params.cs:
          Define la estructura de *Params* en *Action*.
        - Selector.cs:
          Define la estructura del *Selector*.
    - Prop:
        - Prop.cs:
          Define la clase abstracta *Prop*.
        - Faction.cs:
          Define la estructura de *Faction.*
        - Name.cs:
          Define la estructura de *Name*.
        - ParamDecl.cs:
          Define la declaración de los paramteros en *Params.*
        - ParamValue.cs:
          Define la relación parámetro-valor en *Effect*.
        - Power.cs:

Define la estructura de *Power*.
        - Predicate.cs:
          Define la estructura de *Predicate*.
        - Range.cs:
          Define la estructura de *Range*.
        - Single.cs:
          Define la estructura de *Single*.
        - Source.cs:
          Define la estructura de *Source*.
        - Type.cs:
          Define la estructura de Type.
    - Stmt:
        - Stmt.cs:
          Define la clase abstracta *Stmt*.
        - Block.cs:
          Define la estructura de los bloques de código delimitados por *{}*.
        - Expression.cs:
          Almacena la una *Expr*.
        - For.cs:
          Define la estructura del ciclo *for*.
        - Var.cs:
          Define las variables como statements.
        - While.cs:
          Define la estructura del ciclo *while*.
- IClass:
    - IClass.cs:
      Define la interfaz *IClass* para implementar clases del DSL.
    - Card.cs:
      Implementa *IClass*, es la carta en la que esta basado el compilador.
    - Context.cs:
      Implementa *IClass*, define la estructura de *context* como una clase.
    - Lists.cs:
      Implementa *IClass*, define las propiedades de una lista de cartas.
- ICallable:
    - ICallable.cs:
      Define la interfaz *ICallable* para implementar funciones y métodos.
    - Count.cs:
      Define el metodo *Count* de *Lists* para conocer su cantidad de elementos.
    - DeckOfPlayer.cs:
      Define el metodo *DeckOfPlayer* de *context*.
    - FieldOfPlayer.cs:
      Define el metodo *FieldOfPlayer* de context.
    - HandOfPlayer.cs:

Define el metodo *HandOfPlayer* de context.

- GraveyardOfPlayer.cs:
  Define el metodo GraveyardOfPlayer de context.
- Log.cs:
  Define el método *log* para calcular el logaritmo base *a* de *b, log(a, b)*.
- Max.cs:
  Define el método *max* para hallar el máximo entre dos enteros, *max(a b)*.
- Min.cs:
  Define el método *min* para hallar el mínimo entre dos enteros, *min(a b)*.
- Pop.cs:
  Define el método *Pop* de *Lists*.
- Push.cs:
  Define el método *Push* de *Lists*.
- Rand.cs:
  Define el método *rand* que devuelve un número aleatorio entre a y b, *rand(a, b)*.
- Remove.cs:
  Define el método *Remove* de Lists.
- SendBottom.cs:
  Define el método *SendBottom* de Lists.
- Shuffle.cs:
  Define el método *Shuffle* de *Lists*.

- GameState.cs:
  Contiene el estado del juego en un conjunto de listas de tipo *Card*.
- Parser.cs:
  Implementa el *Parser* explicado.
- Enviroment.cs:
  Contiene un conjunto de datos contenidos en un bloque determinado.
- IVisitor.cs:
  Define la interfaz *IVisitor* del patrón Visitor.
- Interpreter.cs:
  Implementa el interpreter ya explicado, implementa *IVisitor*.
- Run.cs:
  Implementa las llamadas en las diferentes etapas de la compilación e interpretación.

Conclusiones.

El compilador juega un papel fundamental en la integración y ejecución eficiente de las cartas en el motor del juego, el compilador convierte automáticamente las definiciones de cartas del DSL a un formato usable por el motor del juego, garantizando que todas las cartas sigan las mismas reglas y estructura. Esto elimina la posibilidad de errores manuales al crear cartas y asegura consistencia en la experiencia del juego. Al permitir la creación de cartas mediante el DSL, el compilador otorga a los jugadores o diseñadores la posibilidad de personalizar el juego de manera sencilla, añadiendo nuevas mecánicas o personajes que enriquecen la experiencia general del juego.